The Education Column

by

Dennis Komm and Thomas Zeume

ETH Zurich, Switzerland and Ruhr-University Bochum, Germany
dennis.komm@inf.ethz.ch and thomas.zeume@rub.de

Logic for Systems: A Gradual Introduction to Formal Methods

Shriram Krishnamurthi
Department of Computer Science, Brown University
shriram@brown.edu

Tim Nelson
Department of Computer Science, Brown University
timothy_nelson@brown.edu

Abstract

We present a proposal for increasing the accessibility of formal methods to the large number of students who could benefit from it but may not be well-served by traditional introductions. Several principles drive our design: a focus on computer systems as the target of study; immersing in applications before theory; building up from programming knowledge; and the breaking down of big transitions into smaller pieces. Concretely, we have created tools, educational materials, evaluation platforms, and more to implement these ideas. This paper describes our philosophy, learning objectives, and learning progression and provides pointers to our materials.

1 Systems as the Motivation for Formal Methods

When the means of production grow more accessible, the need for verification grows in importance. A person growing their food in their backyard has great control over ingredients, whereas one buying it at a supermarket depends on food safety measures. Likewise, the artisan hand-crafting their program is likely to have a much better understanding of it than a person copying code from StackOverflow, GitHub, or AI. Thus, as it has become easier and easier to obtain and now generate code, the value of formal methods (FM) has correspondingly grown.

Nevertheless, we believe that the *accessibility* of FM to the average computer science student has not increased tremendously over the decades. In saying this, we recognize that many of our readers are teachers of courses or authors of books who strongly believe their work does not meet our description or proves our basic premises wrong. We are, of course, not talking about their courses and books, but those of someone else.

On a more serious note, our goal is not to suggest that our ideas are unique (see, e.g., [5, 10, 11, 15, 19, 20]). Rather, we hope to see more of a movement of people who are rethinking approaches to teaching FM. Inasmuch as we share a vision, we would love to hear from you and to work collaboratively! The more materials we generate, the better our examples, the more territory they cover, the better for everyone. We are already growing an ecosystem of free materials.

Crucially, we center our teaching of FM around *computer systems*. Our main course is called "Logic for Systems," and covers systems topics such as role-based access control, garbage collection, electronic locks, mutual exclusion, leader election, and so on. For their final projects, students tackle systems[1] they have encountered in other courses, such as: virtual memory and page tables; process and thread scheduling; cryptographic protocols; distributed hash tables; formal grammars; blockchain, authentication, onion routing; network traffic control; algorithms for max-flow, stable matching, and heuristic search; and ownership and borrowing in Rust.

We focus on systems for several important reasons:

- Systems is a rich source of problems that benefit from the application of FM.
- Systems easily illustrate interesting and difficult parts of FM, such as state exploration.
- Using a uniform set of tools to explore a wide variety of systems topics shows the strength of those tools.
- Inasmuch as many computer science students self-identify as "not-theory people," they often gravitate towards building systems.
- Finally, whatever their leanings in computing, a good percentage of our graduates will go on to work on computer systems. Thus, focusing on systems both speaks directly to their careers and gives them ideas for how to port what they learn to their future reality. Our goal is to equip them with basic training before they go on to build tomorrow's digital infrastructure. We want all students to learn how to apply formal approaches to their own areas of interest and expertise.

It is crucial to recognize what this framing is *not*. It is not a classical logic curriculum, which has little purchase with students who don't care much about the mortality of Socrates as an example of a syllogism. It is not traditional "logic

[1]Because we want students to pursue their own interests, we also allow projects that are not traditionally "systems," provided they are sufficiently complex. These have included topics such as cellular automata, group theory, graph theory, election theory (e.g., Arrow's theorem), particle decay, chemical reactions, and music theory.

in computer science," which can often be too "on paper," front-loading the "logic" and back-loading the "computer science." Nor is it logic in the narrow service of proofs about programming languages (even though that is where the first author received his training).

One of us had a revealing moment a few years ago. A group of students (who had studied the material below) was discussing their distributed systems course project in the atrium outside his office. After overhearing some of their discussion, he peeked at the code they were writing. To his surprise (and delight), they were not programming—they were building and analyzing formal models! This is a kind of "Turing test" for a "logic for systems" FM course: when an observer cannot tell whether it is systems or FM that students are discussing.

2 Guiding Principles

Our curricular design is inspired by guiding principles:

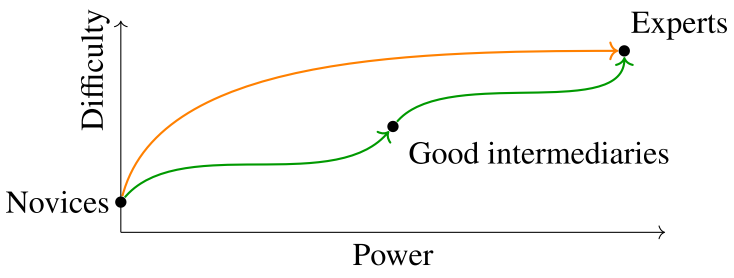
The other 90%. Many university courses are susceptible to the danger of teaching to students who are essentially "mini me"s: i.e., just like the professors. These students love the courses, gravitate to them, engage deeply, do great work, and gush about the offering afterwards, all of which are deeply rewarding to the instructor. Lost in this, however, is a large body of students who are disengaged or simply never take these courses at all.

Our view is that some small percentage—say 10%—fall in the former category, whereas our goal is to design courses to appeal to the other 90%. This is not to say that we want to exclude the 10%! But: (a) that 10% will still benefit from the course, provided it is intellectually honest, (b) they may see the material in a new light, (c) they are more likely to consult classical presentations of the material and learn it by themselves, and (d) many go to graduate school, where they will anyway see the classical version. Whereas *none* of these statements works if we design the course the other way around. Our focus, therefore, is to design FM education for the other 90%.

Play it Backwards. Many traditional "Logic in Computer Science" courses have a familiar progression: from propositional to predicate to first-order logic, and perhaps beyond; covering consistency, completeness, compactness, and other c's; and often ending with some demonstration of tools that implement these logics. These courses are excellent for the mathematically-minded—we were mini-me's who enjoyed these courses ourselves. But they fail to connect with the other 90%.

Our response is essentially to play this script "backwards." Why does logic matter in computer science? Because we can model systems and prove properties about them, and can do so at scale, using tools. We therefore *start* with tools and model systems. Students do so with a loose but sufficient understanding of the logics they are using (much as they do with the programming languages they use), which are arranged in a careful progression (Section 4). By doing this, we can quickly get to the reason these methods matter. Then, once students have accomplished enough with the tools, we introduce the formal aspects. Because, for instance, students have made extensive use of solvers with finite bounds on universes, they are now *motivated* to care about the theory: Why do those bounds exist? What happens in their absence?, and so on, turning those dry theoretical questions into ones that they can viscerally feel.

Finding "Mid"Points. It is common in FM to find a trade-off between power and difficulty: the more powerful a tool is, usually, the more difficult it is to use. Our job as FM educators is to take novices from points of low power and (relative to the instructors, though perhaps not for the novices themselves!) low difficulty to the position of experts, who have mastered the use of high-powered but difficult tools:



When confronted with these kinds of steep curves and long arcs, our instinct is to ask, "What points can we find in-between that break up this transition?" By finding meaningful "value adds" in-between, we can reveal the full complexity (and richness) of FM more gradually, and bring more students along on the ride. (Notably, our course is *optional*; students must choose to take it. This forces on us a useful discipline of accessibility.)

Combining these principles, we advocate the notion of *gradual* formal methods: a process to *meet students where they are*, with the goal of taking them to where we would like them to be (which, eventually, is where *we* are—and beyond). We are inspired by the Jackson and Wing notion of "lightweight" formal methods [8], but being "gradual" (inspired by the term "gradual typing" [17]) is more than that.

Lightweightness emphasis tractability and automation over completeness, but still sits squarely inside FM. Gradualism extends earlier and draws students into FM in a series of steps.

Although the remainder of this paper presents one possible instantiation of our principles, and the one we feel is best in our context, *we stress that these design principles are not tied to any specific tool.* A course using TLA+ [9], for instance, can also meet our criteria. Indeed, it is harmful educationally that we focus so much on languages and tools rather than on learning outcomes and progressions.

3 Learning Outcomes

After completing our curriculum, we want students to be able to:

- articulate and check correctness properties for their code;
- understand the syntax and semantics of propositional, first-order relational, and linear-temporal logics and identify where each is appropriate to use;
- use these logics to specify states and discrete-event transition systems;
- relate these abstract modeling ideas to real systems like physical locks, automated memory management, and mutual exclusion;
- design and build formal models of systems they have encountered outside the course;
- use model finders, SAT solvers, and SMT solvers to solve constraints;

and some other topics (like implementing a SAT solver and validating proof trees).

4 A Learning Progression

Learning outcomes provide a specification; the following learning progression is one implementation that we believe satisfies it. We evaluate these outcomes not only via performance on assignments, but also in live demos where students present and answer questions about their models, as well as through research and assessment projects.

4.1 From Tests to Properties

We begin by assuming no more background than programming and standard data structures and algorithms, corresponding roughly to CS1–2 courses. We do not assume that they have had a particular kind of programming (e.g., deeply-recursive functional programming), nor much depth in any particular programming language.

These assumptions are partly an artifact of the rest of our curriculum, but they are also useful in making our ideas more broadly applicable.

We assume that students recognize the value of *testing*. In our experience, this is one topic that requires little motivation: even students who refuse to acknowledge its value from classes do so quickly once they have had an industrial internship, and then pass the word along (e.g., as teaching assistants) to those younger than them. It is important that this is uncontroversial, because we build atop this foundation.

What is the correct role of testing with respect to FM? We all know the Dijkstra quote [3] that "Program testing can be used to show the presence of bugs, but never to show their absence." However, while as a method of proof it may be lacking, it is still a method of *specification*. We view tests as a form of specification: rather than conventional specifications, which are abstract (and hence define the system "from above"), tests are *point-wise* specifications that define the system "from below." That is, the weakness of tests from an FM perspective is not the tests themselves, but rather the conventional means of establishing them.

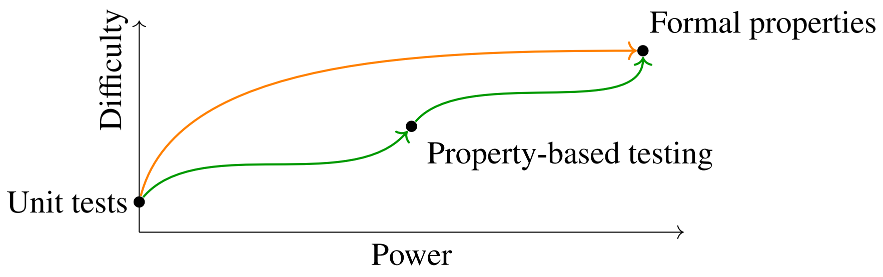
This perspective is important because, rather than being dismissive of testing, we embrace it as an aspect of FM that students are already aware of and comfortable doing. But now we set them up for failure!

Suppose one writes a test that `X must be Y` (where `X` is some program operation and `Y` is its expected output), and it fails. Why might it have failed? The most common reason is because the program is incorrect: `X` has a bug, so it does not produce `Y`. Much less frequently, `Y` is wrong.[2] But can it somehow be that neither is wrong, and yet the test fails?

An underlying assumption behind traditional unit testing is that the computation is a function: for a given input, it produces the same output. In situations where it is imperative, of course, the tester needs to do "setup" and "teardown" work to get the system into the right configuration (and then restore it). However, there is a more subtle case that lacks this input-output simplicity: when the "function" is actually a *relation*. In that case, one could have written a perfectly valid output (`Y`), but it happens to not correspond to what *this* implementation produces (*now*).

There are, in fact, many problems encountered regularly in computing that have this relational nature, which students can encounter relatively early in their education. For instance, classical graph algorithms—shortest paths, minimum spanning trees, etc.—can have multiple results, and which one is produced is clearly a function of internal implementation details. By running student tests against multiple correct implementations, each with slightly different internal strategies, we can expose the brittleness of unit tests.

[2]The reasons for this failure are important. Often, it is due to a typo or light misunderstanding. But sometimes, it can reflect a significant misunderstanding of the problem itself! Some pedagogic curricula are able to exploit this difference: [16] summarizes this line of work.



Clearly, then, a proper test needs to check for membership in a set of answers, or alternatively check for the "shape" of the answer. The former may be feasible when a given input only has a small number of outputs, but is clearly impractical in general. Instead, students learn to write recognizers of answers. In other words, they learn *property-based testing* (PBT, an old idea, but in its modern incarnation most closely associated with the QuickCheck project [2]), and through the very practical expedient of making robust unit tests, have learned to move from individual tests to *specifications*.[3]

We have written two detailed papers [14, 21] on teaching PBT, with several problem examples, as well as a detailed methodology of how to evaluate student work. An added side-effect of this work is that it sets students up to think relationally, which will become important later.

4.2 From Programs to Specifications

Students have now written specifications of the *properties* that should hold of their implemented systems, using a familiar programming language to do so. The next step is to specify the *systems* themselves formally.

Given our goals (Section 2), we used Alloy [7] because of its automation, and a syntax that mimics the object-oriented programming languages that most students have already seen. Moreover, Alloy's relational language is well-suited to modeling object relationships [6, 18] and concepts such as non-determinism, making it a great fit for modeling systems. Finally, Alloy is fundamentally grounded in model-*finding* (through satisfiability). This is crucial: in addition to supporting traditional verification against properties, it also enables property-*free* exploration of models, which addresses the oft-overlooked question of where properties come from in FM.

[3]We note an added benefit to this approach. Once students have been exposed to computational complexity—e.g., big-O—we can now discuss the complexity of a solution versus that of the recognizer. With an appropriate choice of examples—we have found subset-sum and Hamiltonian circuit especially useful—they recognize that there are problems where checking the solution is clearly in polynomial time, even though they cannot see any tractable way to solve it in less than at least exponential time. In short, this serves as an elegant and well-motivated introduction to NP-completness, leveraging their experience with PBT.

```
sig Counter {var value: one Int}
pred someTrace { Counter.value = 0 and
  always { Counter.value' = add[Counter.value, 1] } }
```

Figure 1: Modeling a counter in Temporal Forge. The syntax is similar to Alloy 6: each counter atom contains a value that varies over time. The system advances the counter by 1 with every transition.

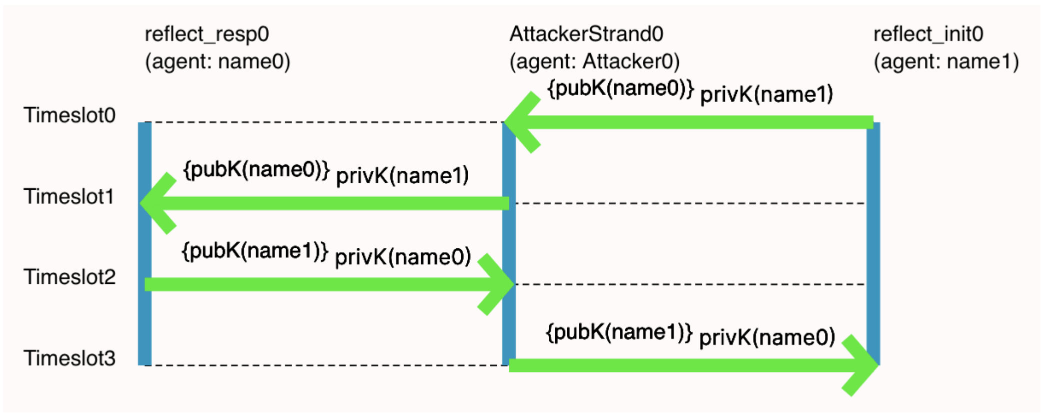


Figure 2: A domain-specific visualization of a cryptographic protocol execution.

By exploring system configurations, students can inductively identify undesirable states and turn these into formal properties to verify. This gives students a much richer and more accurate view of FM than just "verification": as all serious FM practitioners know, its value lies as much in formalization as in checking, but this is a lesson that can only be internalized from experience, not from lecture.

We now use Forge [13], a pedagogic variant of Alloy. Forge inherits all the qualities described above. In addition, it provides support for *domain-specific visualization*, *domain-specific input languages*, and a *nested series of specification languages* that gradually add power to what students can specify. Crucially, each language is self-contained, and not just a fragment of some larger language. Each has its own epistemic closure [4], coming equipped with its own (e.g.) error messages that limit themselves to concepts covered at that language level. Figures 1 and 2 show a basic example of Forge syntax and a domain-specific visualization for a more complex model, respectively.

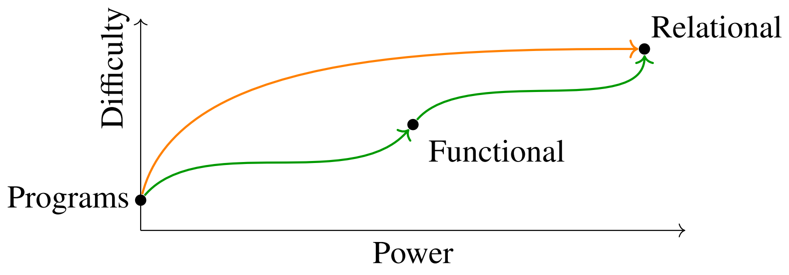
Forge is open-source and available for free at www.forge-fm.org. In addition to all the features described in this paper, the tool comes with a variety of supplemental materials and a draft textbook, making it ready for immediate use.

4.2.1 Starting with Functional Specifications

The language progression begins with *Froglet*, a subset of Alloy that allows specification with only first-order partial and total functions. For all its power, relationality comes at a cost: a larger vocabulary of operators, more subtle behavior (especially around joins), and so on increase the cognitive load of learning to model systems correctly and debug the solver output from a buggy specification. Even students who have taken a discrete math course and learned about relations in a mathematical context (which not all of our students will have done) will not necessarily be fully prepared for the visceral experience of actually using relations "for real."

Students are already used to the dotted field navigation syntax of objects and structures. Because Froglet uses the dot operator as function application (and not as general relational join, as Alloy does), students can describe constraints in a familiar way, and focus their learning on central concepts like quantification, using a pre-and-post-state pair in specification, and so on. Combining these concepts with the familiar dot-as-application notation can be surprisingly powerful. For example, if a student is modeling a leader election (perhaps as part of a protocol like Raft) they might allow a server to step down as leader if another server is more up-to-date: `some s2: Server | s2 != s and s2.currentTerm > s.currentTerm`. After some homeworks, students undertake a midterm project in which they must choose a system to model and then reason about it using techniques like invariant checking (with initiation and consecution, à la Manna and Pnueli [12]) or bounded model-checking [1].

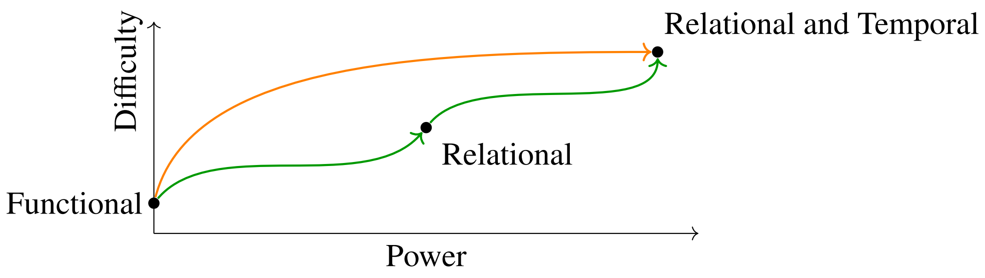
4.2.2 From Functional to Relational Specifications



Of course, not all object relationships are ideally specified as functions. Even the basic concept of a directed graph, while technically possible to model as a Boolean-valued function, can be worked with more conveniently as a relation. When students advance to *Relational Forge*, they gain access to a variety of relational operators, such as transitive closure and join. Students learn that the partial functions of Froglet are just constrained relations and (following Alloy) that the familiar dot operator is useful for far more than just field access.

For example, a student modeling basic network behavior can now represent the forwarding policy as a *relation* on the destination address, input host, and output host (`Address` × `Host` × `Host`), which is natural since a router may send a packet out multiple ports. The student can then use set comprehension and the product (`->`), transitive closure (`^`), and join (`.`) operators to build the host-reachability relation: `{s, d: Host | some a: Address | s->d in ^(a.policy)}` which would not be possible in Froglet. Students first use Relational Forge to specify and reason about automated memory management and mutual exclusion. Many choose to continue using it on their final projects to model graph algorithms, networks, the blockchain, etc.

4.2.3 From Relational to Temporal Specifications



Students have already done a great deal of temporal reasoning in both Froglet and Relational Forge. But this has largely involved safety and not liveness properties. Students could, in principle, encode liveness properties in Relational Forge, but it is subtle. Given a set of linearly-ordered timestamps, existential and universal quantification over that set are roughly analogous to the "finally" (`F`) and "globally" (`G`) modalities from Linear Temporal Logic (LTL). However, to make this work, students need to choose between encoding lasso traces (for infinite-trace semantics) or the special status of a final state (for finite-trace semantics).

For this reason, we transition to Temporal Forge roughly halfway through the course. Like Alloy 6, Temporal Forge enriches the relational language with future- and past-time LTL operators. This makes it easy to express many liveness properties without the added cognitive load of modeling traces themselves.

5 Other Formal Methods

We have presented a somewhat narrow view of FM through the lens of logic. In fact, many more "formal methods" are invaluable to the design, analysis, and implementation of computer systems, such as probabilistic methods and combinatorial optimization. We have excluded them above for the following pragmatic reasons.

First, of the lot, our own expertise lies most in logic. Second, there is enough to do in improving logic education itself, and in the space of one course, we would rather do one thing well than many things poorly. Third, there is already a rich tradition of education in probabilistic methods, which has grown rapidly with advancements in AI, so we feel we have little to contribute there.

In contrast, combinatorial optimization, with a computational focus and declarative specifications, does not appear to be as widespread (at least in the USA). It is another domain where one writes rich, concise specifications that are executed by powerful engines. Indeed, since Forge relies on solvers, there is a natural bridge from our solvers to traditional numeric ones. However, we are fortunate to work in a department with a whole class on the topic, so we simply point our students to take it next (and many do). In its absence, we would likely add some material about the broader space of solver-driven synthesis and analysis.

6 But What About Proofs?

A perhaps startling absence from our discussion above is any mention of formal deductive proof. This omission is not accidental.

There are many formal methods that have enjoyed widespread success in different spheres, from types to model-checking to even parsing. They all have in common a very high degree of automation. Even though they all come with some underlying proof, the process of arriving at that proof is hidden behind the scenes, and is even often buried in metatheory. The details are immensely valuable to the tool producers, but that is not most consumers.

It is worth asking to what extent students *should* care about proofs. Unfortunately, to make proofs tractable and fit within the time bounds of curricula, the examples they tend to see are either not from computing at all, or are so neutered as to be uninteresting. We are reminded of William Scherlis's comment, that it is better to prove "little theorems about big programs instead of big theorems about little programs," and automated methods allow us to focus on the former.

Of course, this can change! One can imagine providing large degrees of automation early on, and developing an entirely different learning progression that gradually tapers the amount of automation. Indeed, with collaborators at Brown, we have been investigating adding a proof-assistant mode that turns the bounded verification provided by Forge into a proper deductive proof. Innovations in AI-driven proof automation (e.g., using language models) can also significantly help (though arguably with very frustrating failure modes). In short, there is much room for growth—but we believe it will also take a while to approach the sophistication of domains that we can cover. (And we would be delighted to be proven wrong!)

Ultimately, we think an over-emphasis on proofs shuts the door on FM too soon. FM should first prove itself accessible and relevant, through positive examples and not just through scare stories. Once a student has seen the value, an entire landscape of FM—including the many wonderful proof-assistants on offer—lies before them: all our work may merely be a midpoint itself. We should orient the community around gradually ushering the other 90% of students into that world through the power of persuasive examples.

Acknowledgments

We are grateful to the generations of students who have worked through iterations of this material. We appreciate our many co-authors who have built the tools and ideas that we describe here. We thank Kathi Fisler both for collaborating on this path over several years and for several comments on a draft, and Serdar Kadıoğlu and Rob Lewis for enlightening conversations. Matthias Felleisen exposited language levels as a discrete step function, which we have reformulated as a search for midpoints. Finally, we thank Dennis Komm and Thomas Zeume for editorial work.

References

[1] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1999.

[2] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*, 2000.

[3] E. W. Dijkstra. Structured programming. In *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee*, 1970.

[4] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[5] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

[6] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[7] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2nd edition, 2012.

[8] Daniel Jackson and Jeanette Wing. Lightweight formal methods. *IEEE Computer*, April 1996.

[9] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[10] Konstantin Läufer, Gunda Mertin, and George K. Thiruvathukal. Engaging more students in formal methods education: A practical approach using temporal logic of actions. *Computer*, 57(12):118–123, December 2024.

[11] Nuno Macedo, Alcino Cunha, José Pereira, Renato Carvalho, Ricardo Silva, Ana C. R. Paiva, Miguel Sozinho Ramalho, and Daniel Castro Silva. Experiences on teaching Alloy with an automated assessment platform. *Sci. Comput. Program.*, 211:102690, 2021.

[12] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems*. Springer, 1995.

[13] Tim Nelson, Ben Greenman, Siddhartha Prasad, Tristan Dyer, Ethan Bove, Qianfan Chen, Charles Cutting, Thomas Del Vecchio, Sidney LeVine, Julianne Rudner, Ben Ryjikov, Alexander Varga, Andrew Wagner, Luke West, and Shriram Krishnamurthi. Forge: A tool and language for teaching formal methods. In *Object-Oriented Programming Systems, Languages, and Applications*, 2024.

[14] Tim Nelson, Elijah Rivera, Sam Soucie, Thomas Del Vecchio, John Wrenn, and Shriram Krishnamurthi. Automated, targeted testing of property-based testing predicates. In *The Art, Science, and Engineering of Programming*, 2022.

[15] Doron A. Peled. *Software Reliability Methods*. Springer, 2001.

[16] Brown PLT. The Examplar project: A summary. https://blog.brownplt.org/2024/01/01/examplar.html, 2024. Accessed: 2025-02-23.

[17] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming*, pages 81–92, September 2006.

[18] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[19] Maurice ter Beek, Manfred Broy, and Brijesh Dongol. The role of formal methods in computer science education. *ACM Inroads*, 15(4):58–66, November 2024.

[20] Maurice H. ter Beek, Rod Chapman, Rance Cleaveland, Hubert Garavel, Rong Gu, Ivo ter Horst, Jeroen J. A. Keiren, Thierry Lecomte, Michael Leuschel, Kristin Yvonne Rozier, Augusto Sampaio, Cristina Seceleanu, Martyn Thomas, Tim A. C. Willemse, and Lijun Zhang. Formal methods in industry. *Form. Asp. Comput.*, 37(1), December 2024.

[21] John Wrenn, Tim Nelson, and Shriram Krishnamurthi. Using relational problems to teach property-based testing. In *The Art, Science, and Engineering of Programming*, 2021.