Usability and Adoption of Formal Verification Through the Lens of Dafny

Anna Gerchanovsky

COMSCI 587 Fall 2025

1 Introduction

My goal with this work is to cover the usability and adoption of formal verification through the lens of the programming language and automated verifier Dafny. This paper covers a survey of existing literature and my own experience. I begin my covering my own experience with Dafny - discussing my previous experience with the language, particularly in the computer science education setting, and covering my experience working on some simple Dafny examples for this paper. I next cover how Dafny fits into computer science education as a whole, focusing on how it addresses existing challenges in teaching formal methods and what challenges either persist or emerge when using Dafny. I take the use of Dafny in computer science education as a motivating example as it emphasizes how usable and understandable Dafny is and any issues that emerge in a classroom setting when using Dafny are likely to emerge again in more complex use-cases. I finally cover the usage of Dafny in general, beyond computer science education, and find that many issues that emerge in my experience and in computer science education are seen again and are being addressed through various extensions, additions, or augmentations to Dafny.

1.1 Dafny Basics

Dafny is a programming language and automated program verifier introduced by K. Rustan M. Leino in 2010 [1]. Dafny code is translated to the verification language Boogie, which then generates conditions to be verified by an SMT solver.

Dafny has support for the following features: [2]

1. **Types:** Dafny includes the types `int` (integers), `bool` (boolean), `seq<T>` (immutable arrays with elements of type `T`), `array<T>` (mutable arrays of type `T`). Dafny has support for user defined types as well.

2. **Methods:** In Dafny, a `method` has input values and output values, with given names to return values.

3. **Functions:** In Dafny, a `function` is an expression that takes input and returns its body.

4. **Predicates:** A `predicate` is a function that has return type `bool`.

5. **Lemmas:** Lemmas, formerly known as `ghost methods`, do not modify the input and instead prove a property of it.

6. **Preconditions:** Preconditions are defined by `requires` statements.

7. **Postconditions:** Postconditions are defined by `requires` statements.

8. **Loop Invariants:** Loop invariants are defined with `invariant` statements.

9. **Additional Verification:** Additional verification is done with `assert` statements or by calling a `lemma` with a given postcondition.

Below is a small example some of these concepts.

```
predicate isNonNegative(a:int)
{ a ≥ 0 }

function increment(a:int):int
{ a + 1 }

method returnVal(a: int) returns (b: int)
  requires isNonNegative(a)
  ensures a == b
{
  b := 0;
  while (b < a)
    invariant b ≤ a
  {
    b := increment(b);
  }
  /* b is returned, as specified in the method header */
}
```

Listing 1: A basic Dafny code example using a predicate and lemma to supplement a method

Dafny supports integrated verification through IDEs that verify code as it is written, highlighting invariants, assertions, or conditions that may not hold, or at least ones that the verifier cannot confirm will hold [3].

Software verification with Dafny is meant to be accessible and easy [4]. Examples of Dafny verification are published in papers and online [2, 5]. Leino's *Program Proofs* is a textbook that covers writing and satisfying program specifications using Dafny [6].

1.2 Motivation

My motivation in exploring the adoption of formal verification, in particular in computer science education and in particular using Dafny, was in part personal. My first introduction to pre and post conditions was in my introductory imperative computing class in undergrad, where we used a variant of C called C0 which included "contracts", which functioned similarly to pre and post conditions, loop invariants, and assertions, and were checked at runtime [7]. Writing useful and correct contracts was required in the course. To my understanding, most

students were relieved when, at the end of the course, instruction moved to standard C and contracts were no longer required.

Later, I was introduced to Dafny in my introductory computer security course. In this course, we used Dafny to verify simple programs, like going through an array and doubling every element in it. I was already familiar with pre and post conditions, loop invariants, and assertions, but, nontheless, getting these programs to work and verify posed a surprising challenge. However, going through this process - adding correct invariants, adjusting my code, figuring out edge-cases - truly helped me better understand how to build proofs and understand what ensuring correctness meant practically.

I went on to serve as a teaching assistant for this iteration of the course for five semesters, during which Dafny was at times omitted from the curriculum in the interest of time. The general sentiment was that Dafny was interesting and impressive, but difficult and a bit too isolated from the rest of the course. Using Dafny in recitations proved frustrating to students. Seeing their correct program fail to verify while an equivalent program succeeded due to some logic they could not see or explain made it difficult to fully understand or enjoy the process - and that was if students were able to successfully able to install and use Dafny on their code editors in the first place.

During this time, I took a graduate level computer security course which featured Dafny more heavily. In this course, our final task was to implement a simple language that satisfies type safety and non-interference in Dafny. This was, by far, my most difficult experience using Dafny. I worked in a group of three, and it took us weeks and hundreds of lines of code to implement. I came away from this with a new, much more frustrated but much more impressed perspective on Dafny than the previous play examples.

In "Program Proofs", K. Rustan M. Leino says the following:

> *"When I first learned about program verification, all program developments and proofs were done by hand. I loved it. But I think I was the only one in the class who did." (page IV)* [6]

To some extent, I feel the same way. I have always enjoyed writing mathematical proofs and the logic behind them. Any frustration I felt using Dafny, during my work on this project or any time before, was tempered by my interest in program verification and my immense satisfaction when I was able to get the "puzzle" of getting Dafny to verify. I know however, from my peers and from my experience as a teaching assistant, that many computer scientists do not feel the same way. One of the research areas I am most interested in is usable security. Understanding then both the value of using Dafny and assuming that many Dafny users will have a different (probably less enjoyable) experience than I did, I am curious what the current usage of Dafny looks like in general.

I wanted to return to the subject of Dafny for this project. My hope, in this project, is to see how universal my experience was with Dafny, explore the current research on its benefits, and see if anything has changed.

1.3 Goals

In this paper, I aim to address the following questions:

RQ1) What is the experience of using Dafny to write code? What are the benefits and difficulties? This is addressed in §2.

RQ2) How can using Dafny help address some of the challenges in teaching formal methods? What difficulties persist when using Dafny? This is addressed in §3.2.

RQ3) What challenges come up in general when using Dafny, and are there solutions to these challenges? This is addressed in §4.

2 My Experience

In writing this work, I experimented with implementing some simple algorithms in Dafny, without referencing existing examples. My goal was to note my own difficulties or challenges, as well as positive observations, that could occur when writing novel code in Dafny.

2.1 Example 1: Get Largest Value

The first algorithm I attempted implementing was a method that would return the largest value in an array.

```
0  method getLargest(a: seq<int>) returns (x: int)
1      requires |a| > 0
2      ensures ∀ i :: 0 ≤ i < |a| ⟹ a[i] ≤ x
3  {
4      var i := 0;
5      x := a[i];
6      while i < |a|
7          invariant i ≤ |a| /* necessary for next invariant */
8          invariant ∀ j :: 0 ≤ j < i ⟹ a[j] ≤ x
9      {
10         if a[i] ≥ x {
11             x := a[i];
12         }
13         i := i + 1;
14     }
15 }
```

Listing 2: An implementation of `getLargest`

In this example, the invariant on line 8 is necessary for the next invariant on line 9 to hold. This invariant is very redundant, but is necessary for the program to verify. Additionally, the invariants on line 8 and 9 cannot be switched in order.

I expect that coming up with this invariant would be difficult for a student learning Dafny. It is reasonable to expect for this invariant to be unnecessary. Even when a student writes this invariant, they may first think to write `invariant i < a.Length`, mirroring the loop guard. Although this is incorrect, since invariants must hold before and after any iteration of the loop, writing a loop invariant that is strictly weaker than the loop guard *directly* above it can be a difficult concept to understand.

2.2 Example 2: Get 2nd Largest Value

The next algorithm was a slight extension to the previous example - returning the second largest element of an array. While very similar to the previous example, even defining the scope of this method (does there need to be at least two different values in the array, therefore requiring the array to have length of at least 2? or is there a defined action if the array only has one unique element?) becomes more difficult. This is actually, in my opinion, a very helpful feature of Dafny. By forcing the programmer to consider the cases in which the method will work, Dafny can build a habit that can be applied to traditional languages. I was forced to consider for what cases this method will work for.

For my implementation, I decided there had to be at least two unique elements in the array. This only required adding the pre-condition

```
0 requires ∃ i, j :: (0 ≤ i ≤ j < |a| ∧ a[i] ≠ a[j])
```

The post-condition was much more difficult. My initial idea was the following:

```
0 ensures ∀ i :: 0 ≤ i < |a| ⟹
1     (
2         (∀ j :: 0 ≤ j < |a| ⟹ a[j] ≤ a[i])
3         /* a[i] is the largest element */
4         ∨
5         (a[i] ≤ x)
6         /* a[i] is smaller than the 2nd largest element x */
7     )
```

So, for any element `a[i]`, it is either the largest element (line 3), or it is less than or equal to the return value `x` (line 4). However, this pre-condition does not actually gurantee desired behavior. This method could still return the largest element. In order to fix this, another post-condition must be added.

```
0 ensures ∃ i ::
1     (
2         (0 ≤ i < |a|) ∧ /* it is a valid index */
3         (∀ j :: 0 ≤ j < |a| ⟹ a[j] ≤ a[i]) ∧
4         /* a[i] is the largest element */
5         (a[i] > x)
6         /* a[i] larger than the 2nd largest element x */
7     )
```

This code becomes bulky, encouraging the use of predicates, even for a very simple method.

```
0  predicate hasMultipleValues(a:seq<int>)
1  { ∃ i, j :: (0 ≤ i ≤ j < |a| ∧ a[i] ≠ a[j]) }
2
3  predicate isElement(a: seq<int>, x:int)
4  { ∃ i :: (0 ≤ i < |a|) ∧ a[i] == x }
5
6  predicate isLargest(a: seq<int>, x:int)
7  {
8      ∀ i :: 0 ≤ i < |a| ⟹ a[i] ≤ x
9      ∧ isElement(a, x)
10 }
11
12 predicate isSecondLargest(a: seq<int>, x:int)
```

```
{
    ∀ i :: 0 ≤ i < |a| ⟹ (a[i] ≤ x ∨ isLargest(a, a[i]))
    ∧ !isLargest(a, x)
    ∧ isElement(a, x)
}

method getSmallest(a: seq<int>) returns (x: int)
    requires |a| > 0
    requires hasMultipleValues(a)
    ensures isSmallest(a, x)
{
    ... /* see getLargest */
}

method getSecondLargest(a: seq<int>) returns (x: int)
    requires |a| > 0
    requires hasMultipleValues(a)
    ensures isSecondLargest(a, x)
{
    var y := getLargest(a);
    var i := 0;
    x := getSmallest(a);

    while (i < |a|)
        invariant isElement(a, x)
        invariant 0 ≤ i ≤ |a|
        invariant x < y
        invariant ∀ j :: 0 ≤ j < i ⟹
            ( (a[j] == y ∧ a[j] > x) ∨ a[j] ≤ x )
    {
        if a[i] ≥ x ∧ a[i] < y {
            x := a[i];
        }
        i := i+1;
    }
}
```

Listing 3: An implementation of `getSecondLargest`

2.3 Takeaways

My implementation is certainly not the shortest or simplest possible. I have not used Dafny in years and my Dafny experience has always been limited, but I am still familiar with Dafny and formal verification. Nonetheless, this simple method took me multiple attempts over multiple hours to implement, which genuinely surprised me.

There were multiple valid solutions that I moved away from because they required more verification code (for example, I set `x := getSmallest(a)` rather than `x := a[0]` so the loop invariant `x < y` would hold). The resulting code is less efficient (although not by much).

At the same time, this process actually helped me find a bug. My first idea for an implementation of `getSecondLargest` was the following:

```
method getSecondLargest(a: seq<int>) returns (x: int)
```

```
1       requires |a| > 0
2       requires hasMultipleValues(a)
3       ensures isSecondLargest(a, x)
4   {
5       var y := a[0];
6       var i := 0;
7       x := a[0];
8
9       while (i < |a|)
10      {
11          if a[i] > y {
12              x := y; /* set x to the previous largest element */
13              y := a[i];
14          }
15      }
16  }
```

Listing 4: An incorrect implementation of `getSecondLargest`

This method is incorrect, and I realized this in the process of attempting to write loop invariants that would allow the program to verify. For the sequence `[5, 4, 3, 2, 1]`, for example, the conditional on line 12 would never hold and the method would return `5`, the largest value.

Additonally, in developing `getSecondLargest`, I was forced to add predicates and make my code more modular. The predicate `isLargest` should have been used in `getLargest` as a post-condition. I found that I needed to add the predicate `isElement`. This was actually missing from `getLargest`! A method satisfying the post-condition for `getLargest` could return the maximum element of `a` plus one.

Part of my takeaways was certainly how much effort using Dafny can be. However, more importantly, I came away from this experience with a greater appreciation for Dafny. As frustrating as the experience of writing and rewriting a simple method in Dafny was, ultimately, without Dafny, I may not have noticed the error in my original method.

3 Computer Science Education

In this section, I will lay out the reported difficulties in teaching formal methods. I will then move onto results of using Dafny in computer science education, as well as the success or limitations Dafny has in addressing these challenges.

3.1 Challenges in Teaching Formal Methods

Within the body of literature focused on teaching formal methods, there is a focus on getting around the difficulties of teaching this subject. "Teaching Formal Methods in Academia" organizes its literature review around what the difficulties in teaching formal methods are and what strategies are used to alleviate these difficulties [8]. Another paper, "Introducing Formal Methods to Students Who Hate Maths and Struggle with Programming", focuses on circumventing difficulties have with formal methods - as is clear from the title - but

nonetheless takes an optimistic view, stating that despite the difficulties formal methods can be taught as early as high school [9].

Many of these papers echo the same difficulties students have with formal methods, including:

1. **Difficulty with Mathematical Concepts:** Students may either lack or struggle with mathematic concepts used when teaching formal methods [8, 9]. Even in cases where students are capable of using the mathematical concepts, their difficulty can drive students away from chooosing formal methods courses [10]. Solutions have included omitting as much math as possible when introducing concepts [9, 11].

2. **Percieved Unimportance of Material:** Formal methods can be isolated from other courses and, without due focus on the importance of thier application, students can struggle to find interest [8]. Solutions include explaining real life applications of formal methods - e.g. saftey properties of a metal press [9].

3. **Lack of Feedback:** Students who are comfortable with programming can struggle with the theoretical nature of formal methods. The lack of immediate and clear feedback in some formal methods tools can be frustrating for students [8].

As we see in future sections, many of these difficulties can be addressed, at least in part, by using Dafny.

3.2 Application of Dafny

Since its introduction, Dafny has been used in teaching formal verification on the graduate and undergraduate level [10–16]. Some courses are taught entirely in Dafny while some only spend a lecture on the material. Multiple cases report high student satisfaction with the course material [10, 14, 15]

The creator of Dafny, K. Rustan M. Leino, incidates that Dafny is an appropriate tool for learning formal verifcation. His book *Program Proofs*, in which Leino includes a note for teachers, is instructional in tone and focuses mostly on Dafny [6].

In an analysis of common algorithms (e.g. insertion sort, priority queue, integer division) that would reasonably be covered in a class introducing Dafny, the amount of verification code was found to be comparable to the amount of implementation code (there were 1.14x as many lines of verification code as there was implementation code), though this varied by algorithm [14]. This indicates that, in the context of computer science education, usind Dafny may impose an overhead that doubles the amount of lines of code necessary.

3.2.1 Benefits

In "Learn'em Dafny", James Noble proposes using Dafny in a formal verification course [11]. Noble suggests a "programming-first" approach, allowing students to focus on coding and allowing the formal verification to follow. This mirrors the suggestions from other work to omit unnecessary mathematics at first in order to make formal verification more approachable and focus on the motivation [8, 9].

Rather than requiring mathematical expertise, Dafny allows students to omit these steps and instead focus on *what* needs to be proved and, perhaps most importantly, *why*, as well as writing the code that can satisfy this. This corroborates my experience working with Dafny for this paper. Through omitting some steps of the proving process, I was able to more quickly write a correct algorithm and prove its correctness, while enforcing good programming practices.

In courses that use Dafny in formal verification instruction, students report Dafny helping them understand pre and post conditions and loop invariants [15] and report that Dafny assessments helping them learn and recieve helpful feedback [10] (addressing the concern of insufficent feedback informal methods).

3.2.2 Limitations

Previous work detailing the experience of instructors working with students has repeatedly demonstrated the gap between understanding or writing simple Dafny code and implementing intermediate-difficulty algorithms in Dafny.

In "Lessons of Formal Program Design in Dafny", Ran Ettinger describes using Dafny in an elective course for advanced undergraduates called "Correct-by-Construction Programming" [16]. The final assessment included three exercises - merge sort, element insert into max-heap, and element insert into a binary search tree. For reference, the course had already covered an implementation of heap sort. Only 8/34 groups were able to submit fully verified solutions for heap insert. In fact, 9 groups had logically incorrect lemma specifications, indicating that, even as Dafny allows a good portion of high-level math to be omitted, a certain level of expertise is still required. Ettinger does note, however, that this exercise was the most difficult out of the three and the fact that there were 8 fully verified groups is still encouraging. Nonethless, this does indicate how much preparation may be necessary in order to use Dafny in more complex applications.

Perhaps more interestingly, a similar pattern is observed in graduate level courses by Faria et al. in "Case Studies of Development of Verified Programs with Dafny for Accessibility Assessment" [14]. In this formal methods course, only students who had performed well, 28/151 total students, were invited to implement a medium complexity algorithm in Dafny given a one month timeline. Examples of valid algorithms include stable marriage and text compression. It should be noted that, unlike Ettinger's course, this course did not use Dafny throughout and did not deeply address advanced verification techniques. Out of 14 students who even *agreed* to take on the challenge, none were able to fully meet the goals laid out.

These results are certainly in line with my own experience. The examples I worked on for this paper are of much lower complexity than these projects, and were still measurably difficult. My previous in-class experience on a more complex Dafny projects I believe only succeeded because we worked in a group, were given an extended period of time, were all familiar with the language beforehand, and were given a very solid foundation of starter code.

While these projects do not indicate that Dafny is any less appropriate for formal methods instruction, they do highlight the difference in difficulty between understanding and implemnting the basics of formal verification in Dafny and applying these concepts to more complex projects, both inside and outside of the classroom.

4 Software Engineering

As Dafny is used in Software engineering environments, the same difficulties as those in §3.2.2 can be reflected. While software engineers may be able to complete the tasks most students fail at, the size of the difficulty gap between proof-of-concept Dafny code and even medium complexity code will affect the dificulty of using Dafny in even higher complexity tasks.

4.1 Observed Difficulties

Ifran et al. describe their experience developing XDsmith, a fuzzer for Dafny, in "Testing Dafny (Experience Paper)" [17]. They report that access to and interaction with Dafny domain experts was helpful in building their understanding of Dafny semantics. They aimed to find comopiler, soundness, and precision bugs. Precision bugs are the most interesting - a correct program can still be rejected by Dafny if its cannot be resolved by the prover. The implementation of `getLargest` in Listing 1.1 without loop invariants, for example, would not verify despite still being correct. So, XDsmith limits itself to finding cases of *correctly annotated* correct programs for precision bugs. XDsmith found 8 precision and 3 soundness bugs in Dafny. The example highligted in the paper was that "`[1]` is a prefix of `[3]`" was `false`, but "`assert !([1]` is a prefix of `[3])`" fails.

However, beyond precision bugs, getting Dafny to verify correct programs is challenging. Because some steps can be resolved by the verifier but some can not, it can be challenging write verifiable progams. In "Free Facts: An Alternative to Inefficient Axioms in Dafny", Tabea Bordis and K. Rustan M. Leino cite "proof brittleness" of SMT solvers [18]. This brittlness means that even small changes like renaming a variable can cause the verifier to take longer to verify or even fail to verify altogether.

In my experience, the invariant below is somewhat redundant.

```
0  method foo(a: int) returns (b: int)
1  {
2      ...
3      assert b ≤ a;
4      while (b < a)
5          invariant b ≤ a
6      {
7          ... /* no operations on b */
8          b := b + 1;
9      }
10     ...
11 }
```

Listing 5: A loop guard related invariant

This invariant should follow from the assertion, loop guard, the incrementation of b by only 1, and the fact that the variables are integers. However, this invariant is necessary in line 12 of `returnVal` (Listing 1.1), line 7 of `getLargest` (Listing 2.1), and line 38 of `get2ndlargest` (Listing 2.2). While this invariant is fairly simple, in some cases writing assertions or invariants that seem redundant while being able to omit more complex steps of the proof can be confusing for programmers.

4.2 Extensions

To help with the issues laid out in previous sections, a collection of extensions or additions to Dafny have been proposed or implemented.

4.2.1 Semantic Extensions

In "Free Facts", Bordis et al. describe programmers adding assert statements of theorems in order to verify progams, e.g. `assert a; assert b;` when $a \implies b$ [18]. Free facts are a proof of concept implementation of adding specific theorems as assumptions to the verifier, eliminating the need to include them in code. This extension shows some promise in reducing proof brittleness.

Another extension to Dafny is "Tacny", a Dafny language program verifier that supports 'tactics' [19]. Tactics encode proof patterns, like proofs by induction, allowing these patterns to be reused and therefore reducing the need for repetitive proof code.

Another area of work is providing better counter-examples for incorrect Dafny programs [20]. Although this does not reduce the work of verifying correct Dafny programs, it can disambiguate between programs that fail to verify because the SMT solver is unable to verify them and those that are incorrect. In my experience, this could have shortened the amount of time it took to go from Listing 2.3 to Listing 2.2.

4.2.2 Integration of AI

Beyond semantic extensions to Dafny, another area of research focuses on the application of AI in assisting with Dafny [21–26].

LLMs have been shown to be able to generate plausible Dafny code, and their performance improves through few-shot prompting, where GPT-4 was able to generate successfully verified and correct specifications [22]. Because Dafny code can be automatically verified, Dafny is a good candidate for training LLMs to write code, as the correctness of before unseen code can be automatically verified [21].

Models can specifically be fine tuned to learn correct annotations - with Llama 3.1 8B being able to succeed in writing correct annotations over 50% of the time [23].

Prompt engineering can increase the correctness of LLM-generated Dafny specifications given specifix natural language requests, with up to 100% syntactic and 96.15% semantic correctness for a single response in some cases (DeepSeek-R1 pass@1 given prompts with syntactic rules) [26]. LLMs were also shown to be able to generate Dafny loop invariants, succeeding up to cases where 5 loop invariants were necessary (99% pass@1 success rate for 1 invariant, 33% pass@1 for 5 invariants) when the best approach from multiple closed source models is used [25].

Writing helper assertions, the use of which free facts aim to reduce, requires both correctly identifying the location of the necessary assertion and the assertion itself. LLM assisted tools are able to correctly write individual assertions over 50% of the time and all missing assertions ¿20% of the time - with accuracy increasing to 66.4% and 33.3%, respectively, given ground truth values of the location of the necessary assertion(s) [24, 27].

Dafny is a good candidate for AI assisted code due to its ability to be automatically verified - generating verifiably correct test datasets, testing predictions, and evaluating outputs

can be automated. In return, the assitance of AI tools in writing specifications or verification statements can be incredibly helpful for the ease of use and therefore adoption of Dafny.

5 Conclusion

Similar difficulties in using Dafny re-emerged in my experience, the experience in using Dafny in the classroom setting, and in literature using Dafny. Writing the correct Dafny code to verify a program can be difficult or time consuming - increasingly so when working with more complex code. Dafny has evolved and continues to evolve as research aims to address these issues and decrease Dafny's difficulty.

Despite any challenges to using Dafny, it remains a valuable tool. Its automatic verifier makes it much simpler to use than manual program proofs. As it is interactive and provides quick feedback, it can be a more engaging method to cover formal methods. It allows users to omit much of the necessary math for formal verification, significantly reducing the mathematical barrier to entry.

References

[1] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *Logic for Programming, Artificial Intelligence, and Reasoning*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370.

[2] K. R. M. Leino, "Dafny quick reference," https://dafny.org/latest/QuickReference.

[3] K. R. M. Leino and V. Wüstholz, "The dafny integrated development environment," *Electronic Proceedings in Theoretical Computer Science*, vol. 149, p. 3–15, Apr. 2014. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.149.2

[4] K. R. M. Leino, "Accessible software verification with dafny," *IEEE Software*, vol. 34, no. 6, pp. 94–97, 2017.

[5] K. R. M. Leino, "Developing verified programs with dafny," *Ada Lett.*, vol. 32, no. 3, p. 9–10, Dec. 2012. [Online]. Available: https://doi.org/10.1145/2402709.2402682

[6] K. R. M. Leino, *Program Proofs*. MIT Press, 2023.

[7] "C0 lang wiki," https://bitbucket.org/c0-lang/docs/wiki/Home, accessed: 2025-12-09.

[8] R. Zhumagambetov, "Teaching formal methods in academia: A systematic literature review," in *Formal Methods – Fun for Everybody*, A. Cerone and M. Roggenbach, Eds. Cham: Springer International Publishing, 2021, pp. 218–226.

[9] N. Yatapanage, "Introducing formal methods to students who hate maths and struggle with programming," in *Formal Methods Teaching*, J. F. Ferreira, A. Mendes, and C. Menghi, Eds. Cham: Springer International Publishing, 2021, pp. 133–145.

[10] J. Noble, D. Streader, I. O. Gariano, and M. Samarakoon, "More programming than programming: Teaching formal methods in a software engineering programme," in *NASA Formal Methods*, J. V. Deshmukh, K. Havelund, and I. Perez, Eds. Cham: Springer International Publishing, 2022, pp. 431–450.

[11] J. Noble, "Learn 'em dafny!" in *51st ACM SIGPLAN Symposium on Principles of Programming Languages*, 2024.

[12] M. Fredrikson, "Introduction to dafny," https://www.cs.cmu.edu/~mfredrik/15414/lectures/04-dafny.pdf, 2016, 15414/15614 Automated Program Verification and Testing.

[13] B. Parno, "Securing software: Languages," https://www.andrew.cmu.edu//course/18-330/2024f/lectures/sw-verification-and-dafny.html, 2024, 15330/18330 Introduction to Computer Security.

[14] J. P. Faria and R. Abreu, "Case studies of development of verified programs with dafny for accessibility assessment," in *Fundamentals of Software Engineering*, H. Hojjat and E. Ábrahám, Eds. Cham: Springer Nature Switzerland, 2023, pp. 25–39.

[15] E. Braude, "Using the dafny verification system in an introduction to algorithms class," 2024.

[16] R. Ettinger, "Lessons of formal program design in dafny," in *Formal Methods Teaching*, J. F. Ferreira, A. Mendes, and C. Menghi, Eds. Cham: Springer International Publishing, 2021, pp. 84–100.

[17] A. Irfan, S. Porncharoenwase, Z. Rakamarić, N. Rungta, and E. Torlak, "Testing dafny (experience paper)," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 556–567. [Online]. Available: https://doi.org/10.1145/3533767.3534382

[18] T. Bordis and K. R. M. Leino, "Free facts: An alternative to inefficient axioms in dafny," in *Formal Methods*, A. Platzer, K. Y. Rozier, M. Pradella, and M. Rossi, Eds. Cham: Springer Nature Switzerland, 2025, pp. 151–169.

[19] G. Grov and V. Tumas, "Tactics for the dafny program verifier," in *Tools and Algorithms for the Construction and Analysis of Systems*, M. Chechik and J.-F. Raskin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 36–53.

[20] A. Chakarov, A. Fedchin, Z. Rakamarić, and N. Rungta, "Better counterexamples for dafny," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 404–411.

[21] C. Yan, F. Che, X. Huang, X. Xu, X. Li, Y. Li, X. Qu, J. Shi, C. Lin, Y. Yang, B. Yuan, H. Zhao, Y. Qiao, B. Zhou, and J. Fu, "Re:form – reducing human priors in scalable formal software verification with rl in llms: A preliminary study on dafny," 2025. [Online]. Available: https://arxiv.org/abs/2507.16331

[22] M. R. H. Misu, C. V. Lopes, I. Ma, and J. Noble, "Towards ai-assisted synthesis of verified dafny methods," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643763

[23] G. Poesia, C. Loughridge, and N. Amin, "dafny-annotator: Ai-assisted verification of dafny programs," 2024. [Online]. Available: https://arxiv.org/abs/2411.15143

[24] Álvaro Silva, A. Mendes, and R. Martins, "Inferring multiple helper dafny assertions with llms," 2025. [Online]. Available: https://arxiv.org/abs/2511.00125

[25] J. Pascoal Faria, E. Trigo, and R. Abreu, "Automatic generation of loop invariants in dafny with large language models," in *Fundamentals of Software Engineering*, H. Hojjat and G. Caltais, Eds. Cham: Springer Nature Switzerland, 2025, pp. 138–154.

[26] Y.-H. Lu, X.-Y. Zhu, W. Zhang, and R. Yan, "Formalizing requirements into dafny specifications with llms," in *Formal Methods and Software Engineering*, É. André, J. Wang, and N. Zhan, Eds. Singapore: Springer Nature Singapore, 2026, pp. 97–116.

[27] E. Mugnier, E. A. Gonzalez, R. Jhala, N. Polikarpova, and Y. Zhou, "Laurel: Unblocking automated verification with large language models," 2025. [Online]. Available: https://arxiv.org/abs/2405.16792