Learn ’em Dafny!*

Anonymous Author(s)

Abstract

Verification tools like Dafny are slowly being adopted in software engineering practice — if not for entire projects, then at least for safety or security critical kernels of large systems. Verification offers a direct to the “essential complexity” of software development – building a correct system — rather than the “accidental complexity” induced by learning how to use a particular language or toolset, with their accompanying quirks, oddities, and idiosyncrasies.

Graduate software engineers need to be exposed to verification tools, and the formal methods, techniques, and concepts underlying those tools, even if only to be prepared when they come across those tools later in their professional careers. Unfortunately, we have found that many software engineering students resist formal methods — whether due to perceived difficulty, suspected practical irrelevance, or overall mathematicity.

In this presentation, we will outline how Dafny can be incorporated into a “programming first” software correctness course. We then reflect on some particular features of Dafny’s design, and hypothesize how relatively small improvements to Dafny could remove some of the accidental complexity which seems attendant with students learning the language, hopefully allowing them to focus further on the essential complexity of verification.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*.

Keywords: Dafny, Dafny, Give me your Answer Do!

ACM Reference Format:
Anonymous Author(s). 2023. Learn ’em Dafny!. In *Proceedings of ACM Conference (Conference’17)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Title note

Conference’17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

> The Toad, having finished his breakfast, picked up a stout stick and swung it vigorously, belabouring imaginary animals. “I’ll learn ’em to steal my house!” he cried. “I’ll learn ’em, I’ll learn ’em!”
>
> “Don’t say ‘learn ’em,’ Toad,” said the Rat, greatly shocked. “It’s not good English.”
>
> “What are you always nagging at Toad for?” inquired the Badger, rather peevishly. “What’s the matter with his English? It’s the same what I use myself, and if it’s good enough for me, it ought to be good enough for you!”
>
> “I’m very sorry,” said the Rat humbly. “Only I think it ought to be ‘teach ’em,’ not ‘learn ’em.’”
>
> “But we don’t want to teach ’em,” replied the Badger. “We want to learn ’em—learn ’em, learn ’em! And what’s more, we’re going to do it, too!”
>
> *The Wind in the Willows*, Kenneth Grahame, [14].

0 Introduction

Formal verification of software systems has been a significant research topic for in computer science for 50 years or more [17]. Tools such as Dafny, SAW, or SPIN are increasingly mature enough to support industrial application [15, 35] but a critical barrier to adoption remains a lack of software engineers trained in their use [13].

As formal tools for software verification have transitioned from an esoteric research topic [26] to a set of increasingly practical tools [24], there has been a corresponding demand in the need for courses to teach this material to students — or at least, if not to *teach* students then to encourage students to *learn* the basic of verification — to expose students to verification, to encourage them to engage with verification tools, to build some level of confidence in undertaking program verification (and of course to lure the best students away from research on scaled stochastic perceptrons and onto topics which do not require supplementary ethical supervision from Amnesty International).

Thus, as formal methods’ industrial use has increased, so has their relevance to education [5, 10, 12, 19]; Zhumagambetov [37] offers a relatively recent systematic literature review. Aceto and Ingolfsdottir [2], for example, have described a recent course at the University of Reykjavik, where students can participate in a three week intensive formal methods course at first year. Yatapanage [36] describes a recent second year course taught at De Montfort University that applied formal methods to concurrent programming — although the paper's title highlights most students' concerns when approaching this topic "*Students Who Hate Maths and Struggle with Programming*". Kamburjan and Gratz [19] showed how a custom interactive proof tool can generate a positive effect on student engagement; Körner and Krings[21] describe how pedagogical changes to inquiry-based learning can support the user of formal tools.

In some ways closest to the approach we present here, Ettinger describes how Dafny has been used for six years at Ben-Gurion University to support teaching refinement-style "correct-by-construction" programming [11], and Blazy describes a similar course based on Why3 [4]. Güdemann describes how verification tools can even support similar learning strategies even in applied computer science courses taught using C [16]. Ábrahám, Nalbach, and Promies taught satisfiability checking remotely during the pandemic [1], and Lecomte [22] describes how B has been used to train software engineers, and how that experience has helped the design and improvement of B tools.

In this presentation we outline an approach to solving this problem, based on using Dafny for Learning and Teaching formal verification in a software engineering course. Our work draws heavily upon the work of Noble and colleagues [31] — in the next section we summarise their "programming first" approach to software verification course design, and then we reflect on how Dafny can support that approach to student learning, and also what Dafny can learn from such approaches.

1 Programming First

Noble et al. [31] describe a common context for teaching formal methods as a component of a more general software engineering or computer science programme. Traditional formal methods courses are structured bottom up — "foundations first". This approach starts by introducing students to propositional and predicate logic, then working up through weakest preconditions to Hoare logics and their application in describing and reasoning about software systems, culminating in pencil-and-paper proofs. While effective in high-status institutions or with highly motivated students, for "the rest of us" this approach is often less appropriate. Engineering students have typically already taken compulsory courses including Boolean algebra and logic (as mathematics) and discrete logic (as physics) during first year: another maths or physics course is unlikely to be popular [30]. Most computer science and software engineering programmes are heavily based around programming; most software engineering majors are keen to take practical elective courses to develop programming skill and experience [30, 32].

In contrast, "programming first" approach [31] aims to work top down: starting with a programming language based tool, and using that high-level tool as a context in which to present the key concepts of software correctness — while offering the majority of students an experience that feels much more like programming rather than like doing mathematics.

1.1 Programming First with Dafny

Although designed for the traditional approach, Leino's Dafny text *Program Proofs* [24] can be adapted for "programming first". Such a course can cover all the "core" features of Dafny circa 2020, i.e. Dafny version 2.3.0, including Dafny methods and classes (imperative, and mutable); functions and inductive datatypes (immutable, finitary); pre and postconditions; predicates (Boolean functions); assumptions and assertions; compiled vs ghost code, well-founded recursion and explicit termination measures, pattern matching, destructors; built-in collections (arrays, sets, maps); loops, invariants, and variants; recursive specifications of iterative programs (including transformations between general recursion, tail recursion, and iteration); and representation invariants for dynamic data structures [31].

Key to adapting *Program Proofs* to a "programming first" approach is that — although the textbook contains two chapters of foundational material — the rest of the text does not depend on the content of either of those chapters. In particular, chapter 2 presents the mathematical foundations of Dafny's program logic, based on Hoare Logic and Weakest Preconditions, and chapter 5 presents the notion of proof and Dafny's constructs (function lemmas, `calc` blocks) that can support programmers in making explicit proofs. Where necessary, Dafny's semantics can be presented informally, without reference the formal definitions. Because Dafny is an implicit verification system, students are not able to see what proofs Dafny's solver many have constructed, so they do not need anything more than a naïe notion of proof.

Dafny's autonomic (what Leino has called "auto-active" [25]) verification is critical to a "programming first" approach, because it mean courses can focus on students learning verified programming, rather than teaching students proof. In Dafny, verification is seamlessly incorporated into development practices, rather than a separate step, and programmers (or students) work in the familiar domain of programs, rather than an unfamiliar domain of proofs. We think of this approach as *implicit* verification where programmers annotate their programs with preconditions, postconditions, variants, invariants, as in Eiffel [28], and do not interact directly with formal models or e.g. proof trees. This is in contrast to *explicit* verification technologies such as Coq [6, 33] where programmers must interact with solvers by directly building proofs and proof trees, potentially even extracting programs from those proofs. Dafny's implicit approach still offers many guarantees: Dafny attempts to prove programs totally correct by default, so recursive methods and loops often require programmers to give variants to prove termination, and loops in particular generally require invariants to prove correctness. Array and pointer accesses typically require invariants, assertions, or preconditions to ensure all accesses are within bounds and variables are initialised and non-null. This means that Dafny programmers (and thus students) interact with Dafny's underlying prover indirectly, at arm's length, in terms of definitions in their programs and constructs in the Dafny language, rather than having to learn explicit representations of proof.

More pragmatically, Dafny offers a number of advantages over more sophisticated tools like Coq [33] or Why3 [4]. First, Dafny offers a concrete, ASCII-compliant syntax — being restricted to ASCII means students should feel some familiarity with the notation: students would not need to learn how to type, let alone pronounce, relatively esoteric characters such as α, δ, or o. Dafny's syntax and semantics being based on C♯ and Java should also be familiar. Students can use the development toolsets they already know, such as VS Code, Eclipse, Git — particularly important for students who need tools such as screen readers, magnifiers, or voice control to complete their work.

1.2 Pedagogy

Pedagogically, a course can rely on Dafny itself to provide students rapid formative feedback — simply by requiring students to submit their solutions via the Dafny verifier. In a very real sense, we are able to leverage the "essential difficulty" of formal verification of correctness — that not only must students implement a correct program, but they must also convince the Dafny prover that their implementation is correct — to aid the students in that task. In simple cases, where students' focus on implementing programs, we can directly supply students with the Dafny specifications and the tool itself will provide feedback: either their program verifies against the specification, or it does not. Where students' focus is on writing specifications, we can allow students to verify their solutions against hidden "oracle" specifications, and again Dafny can check that the students' specifications capture important properties described by the oracles, or more straightforwardly, that the students' specifications and the oracles are mutually consistent.

This means courses can take a "flipped" approach, focused on student learning, rather than a lecture based approach, focused on our teaching [29]. Class meetings are centred around a weekly series of small "mastery" questions about Dafny and verification, served from a simple website. The weekly questions are released at the start of each week, and students may discuss the questions, may work in groups, ask for answers, and make any number of attempts at answering them — but are expected to answer the vast bulk of these questions correctly. Class meetings allow students to discuss any of the questions with the class, and the website lets course staff know which questions students are currently finding difficult. Because of the very liberal rules around answering the mastery questions, we can work out the solution to any weekly question in class, and even demonstrate the correct answer and show it verifying: if students choose not to think and merely copy the provided answer, so be it.

Larger summative assignments also incorporate automated feedback. Students can submit answers to the assignments as many times as necessary: by running each submission through the Dafny verifier, students get immediate feedback about their submission. This feedback is quite terse (just the number of assertions verified, or not verified) because it is not intended to replace students' use of IDEs or to substitute for their own attempts at verification — rather it is so students can judge their progress through the course, and in particular, to know when they have completed each part of each assignment. We are careful to ensure that every important concept required by the summative assignments are covered by weekly questions before the assignment is due. Thus, while we can discuss the summative assignments only in broad outline, we can (and do) refer students to the relevant weekly questions which we can discuss in as much detail and at as much length as necessary.

2 Learning about Dafny

The objective of an academic course is that the students in the course will learn something; but an equally important outcome of a course can be that those teaching the course will learn something things too. In programming courses, especially in interactive tool-centric programming courses, the interactive tool is as much a teacher as the (human) course staff: courses are thus opportunities for tools to "learn" about how they are used, how they are understood, which aspects of their design work well (or otherwise) [22].

2.1 Methods vs Functions

Consider a Dafny a method and function to add two numbers:

```
method addM(a: int, b: int) returns (c: int) {c:=a+b;}
function addF(a: int, b: int): int {a+b}
```

The syntax for declaring the return values are different (**returns** vs :); the syntax for actually returning the results are different; a final semicolon is mandatory in the method and forbidden in the function.

The issue here is that Dafny is **not** an expression language: rather Dafny's underlying conceptual model separates stateless pure functions, and stateful imperative methods, and these are really quite different. The semicolon terminates imperative statements, which is why a semicolon is needed in the method, and why a semicolon is not permitted in the function.

2.2 Ghost Function Method

Dafny's declaration syntax has been changed recently [8]:

Old	New
function	**ghost function**
predicate	**ghost predicate**
function method	**function**
predicate method	**predicate**

The old syntax overloaded **method**, to mean both a method (as against a function) and executable code (that would be compiled) as against verification-only code (that would not be compiled). Verification code can depend on executable code, but executable code may not depend on verification-only code. This overloading was rather confusing, and certainly made it difficult to explain the pure function vs. imperative method distinction. The new syntax is a great improvement: the keyword **ghost** marks out verification-only code; in the absence of a **ghost** qualifier, all functions, methods, and predicates are consider executable only code, and consequently compiled.

2.3 Opaque Methods and Transparent Functions

Methods and functions then perform very differently in the verifier:

```
var m := addM(x,y);
var f := addF(x,y);
assert m =x + y;  //Fails to verify
assert f =x + y;  //Verifies
```

Dafny verifies the assertion on line 4, because functions are incorporated into the verification context. Dafny fails to verify the assertion on line 3, however, because methods are always abstracted by their postconditions, and the declaration of `addM` omits postconditions. There are reasons for these choices, but they do make the language more difficult to learn — especially as declaring `addF` as an **opaque function** rather than just a plain **function** would also prevent line 4 from verifying, unless the function declaration was explicitly annotated with the necessary postconditions.

2.4 If Else If Then Else

Dafny's pure expressions and imperative commands have different syntax for emotionally similar constructs. Imperative code uses **if** without **then** with an optional **else** clause, while pure expressions use **if**…**thenelse**.

```
method schrodingerM(cat: bool) returns (status: string) {
  if (cat)
    { status :="alive"; } else
    { status :="dead"; }
}

function schrodingerF(cat: bool): string {
  if cat then "alive" else "dead" }
```

Here semantic consistency (and perhaps, taking imperative syntax directly from C, and the functional syntax directly from Haskell) has been chosen over syntactic consistency: there's no reason why both syntaxen could not use **then** at least! This design is fine as far as it goes, and explainable phylogenetically, but also causes significant confusion in practice, and makes refactoring code more difficult than it needs to be.

2.5 Overloading Curlies

To see the difficulties with refactoring, imagine editing the `schrodingerF` function to turn it into a method:

```
method schrodingerM3(cat: bool) returns (status: string) {
  status := if cat then { "alive" } else { "dead" };
}
```

Unfortunately this code is not correct, and attempting to compile or verify it results in an error such as "`RHS (of type set<seg<char>>) not assignable to LHS (of type string) Resolver`". The problem here is that even though the body of `M3` counts as a method, the **if**…**then**…**else** nested inside it establishes a (syntactic) expression context. Within expression contexts, curly braces {} are used to delimit sets, rather than for grouping as in method contexts – so `{ "alive"}` is a (singleton) set of strings, rather than just a string.

2.6 Constant Variables

The following code attempts to declare a constant and a variable in three different contexts: inside a class, a module, or a method:

```
class Test {
  var question : string;
  const answer : int;
}
//module level
const answer := 42;
var question := "what_do_you_get";

method test() {
  var question : string;
  const answer : int;
}
```

Compiling this code produces the error that "`fields are not allowed to be declared at the module level`" which is fine, especially as many other languages have such a restriction. Both **var** and **const** declarations work fine inside a class. Inside a method, however, only **var** declarations are permitted, not **const**s: the error message that "`this symbol not expected in Dafny`" only serves to increase the confusion.

2.7 Let Variables

Many of these difficulties we've discussed so far relate to the way the underlying semantic model separating imperative and functional code works in Dafny. To a first approximation, we can explain this using a conceptual model that imperative code requires semicolons to mark state transitions, and can update variables: functional code can read variables but not update them, and cannot use semicolons because functions are evaluated within one temporal heap instant.

Unfortunately, that explanation only holds so far, because Dafny, rather than e.g. importing Haskell's let expression syntax:

```
let x = 7 in x * x
```

to name a subexpression value within a containing expression, Dafny overloads some existing syntax:

```
var x := 7;
x * x
```

Here **var** introduces a **const**ant, the semicolon is purely syntactic as there is no state change, and pedagogically one must divert from phylogeny to apology.

2.8 Mutable Object Structure

Dafny is one of the few tools that can verify programs built from composite structures of mutable objects using class invariants and representation sets. In practice, this requires either explicit definitions of "`Valid`" and "`Repr`" attributes [24] which are verbose and complex, or implicit definitions generated via the "`autocontracts`" attribute [23] which are concise but opaque. Few students were able to use either mechanism effectively. Perhaps by building on work verifying Rust programs, such as Prusti [3] and RustBelt [18], it should be possible to add ownership annotations to fields and parameters, to check those annotations as with Rust's borrow checker [9, 20, 27] and thus extend the implicit definitions already generated by autocontracts.

2.9 Verification Debugging

Much of the work of verifying Dafny programs involves students annotating their code — adding require and ensure clauses and assertions until the verifier has enough information to discharge its proof obligations. Students find this hard because it is not obvious what Dafny "knows" at any given program point: which assertions Dafny is able to prove, which assertions Dafny is able to refute, and which assertions Dafny is unable to answer (i.e. where the prover times out). We also observed cases where Dafny is unable to verify an assertion because it does not have enough information about variable values –— this is particularly prevalent in code where e.g. students have forgotten to write method postconditions, or have not realised a particular postcondition is necessary. This manifests as Dafny being unable to verify an assertion about a method's return value, and simultaneously unable to verify the negation of that same assertion. Even good students find this situation intensely frustrating. Ideally Dafny would be able to give programmers more information about what it knows, e.g. by querying its underlying solver [7].

3 Conclusion

> 116. You think you know when you can learn,
> are more sure when you can write,
> even more when you can teach,
> but certain when you can program.
>
> *Epigrams on Programming*, Alan Perlis, [34].

In this presentation, we've outlined a working hypothesis for the design of a Dafny course. We followed a "programming first" approach, aiming to minimize the amount of explicit theory, explicit proof, and explicit metatheory students needed to understand, to make space for developing informal understandings or expectations for the core activities of verification in Dafny — viz. writing pre- and postconditions, loop invariants, and guide assertions, as required. We've also reflected on some of the more accidental difficulties students found while learning Dafny, and speculated that Dafny may be able to learn from their experience to make learning easier in future. We consider this approach was a success, and hope it may be useful to others planning similar courses.

References

[1] Erika Ábrahám, Jasper Nalbach, and Valentin Promies. 2023. Automated Exercise Generation for Satisfiability Checking. In *Formal Methods Teaching*.

[2] Luca Aceto and Anna Ingólfsdóttir. 2021. Introducing Formal Methods to First-Year Students in Three Intensive Weeks. In *Formal Methods Teaching Workshop*. Springer, 1–17.

[3] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[4] Sandrine Blazy. 2019. Teaching Deductive Verification in Why3 to Undergraduate Students. In *Formal Methods Teaching (FMTea).*

[5] Antonio Cerone and Markus Roggenbach (Eds.). 2019. *Formal Methods – Fun for Everybody (FMFun).* Springer.

[6] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press.

[7] Maria Christakis, K Rustan M Leino, Peter Müller, and Valentin Wüstholz. 2016. Integrated environment for diagnosing verification errors. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 424–441.

[8] Dafny Project. 2023. Quick migration guide from Dafny 3.X to Dafny 4.0. https://github.com/dafny-lang/ide-vscode/wiki/Quick-migration-guide-from-Dafny-3.X-to-Dafny-4.0.

[9] Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muşlu, and Todd W Schiller. 2011. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*. 681–690.

[10] Brijesh Dongol, Luigia Petre, and Graeme Smith. 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.* Vol. 11758. Springer Nature.

[11] Ran Ettinger. 2021. Lessons of Formal Program Design in Dafny. In *Formal Methods Teaching (FMTea).*

[12] João F Ferreira, Alexandra Mendes, and Claudio Menghi. 2021. *Formal Methods Teaching (FMTea).* Springer Nature.

[13] Hubert Garavel, Maurice H Ter Beek, and Jaco Van De Pol. 2020. The 2020 expert survey on formal methods. In *International Conference on Formal Methods for Industrial Critical Systems*. Springer, 3–69.

[14] Kenneth Grahame. 1908. *The Wind in the Willows.* Methuen, London.

[15] Samuel Greengard. 2021. *The Internet of Things.* MIT press.

[16] Matthias Güdemann. 2021. Online Teaching of Verification of C Programs in Applied Computer Science. In *Formal Methods Teaching (FMTea).*

[17] Cliff B Jones and Jayadev Misra. 2021. *Theories of Programming: The Life and Works of Tony Hoare.* Morgan & Claypool.

[18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[19] Eduard Kamburjan and Lukas Grätz. 2021. Increasing Engagement with Interactive Visualization: Formal Methods as Serious Games. In *Formal Methods Teaching Workshop*. Springer, 43–59.

[20] Steve Klabnik and Carol Nichols. 2019. *The Rust Programming Language (Covers Rust 2018).* No Starch Press.

[21] Philipp Körner and Sebastian Krings. 2021. Increasing Student Self-Reliance and Engagement in Model-Checking Courses. In *Formal Methods Teaching (FMTea).*

[22] Thierry Lecomte. 2023. Teaching and Training in Formalisation with B. In *Formal Methods Teaching*.

[23] K Rustan M Leino. 2013. Developing verified programs with Dafny. In *2013 35th International Conference on Software Engineering (ICSE).* IEEE, 1488–1490.

[24] K. Rustan M. Leino. 2023. *Program Proofs.* MIT Press.

[25] K. Rustan M. Leino and Michael Moskal. 2010. Usable Auto-Active Verification. In *Usable Verification Workshop (UV10).*

[26] K. Rustan M. Leino and Greg Nelson. 1998. An extended static checker for Modula-3. In *International Conference on Compiler Construction*. Springer, 302–305.

[27] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. 2010. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 2 (2010), 1–37.

[28] Bertrand Meyer. 2009. *Touch of Class.* Springer.

[29] Militsa Nechkina. 1984. Increasing the effectiveness of a lesson. *Communist* 2, 51 (1984).

[30] James Noble, David J. Pearce, and Lindsay Groves. 2008. Introducing Alloy in a Software Modelling Course. In *1st Workshop on Formal Methods in Computer Science Education (FORMED).*

[31] James Noble, David Streader, Isaac Oscar Gariano, and Miniruwani Samarakoon. 2022. More Programming Than Programming: Teaching Formal Methods in a Software Engineering Programme. In *NASA Symposium on Formal Methods*.

[32] Aaron Pang, Craig Anslow, and James Noble. 2018. What programming languages do developers use? A theory of static vs dynamic language choice. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, 239–247.

[33] Christine Paulin-Mohring. 2011. Introduction to the Coq Proof-Assistant for Practical Software Verification. In *LASER International Summer School*. Springer.

[34] Alan Perlis. 1982. Epigrams on programming. *SIGPLAN Notices* 17, 9 (Sept. 1982).

[35] Hillel Wayne. 2018. Temporal Logic. In *Practical TLA+*. Springer, 97–110.

[36] Nisansala Yatapanage. 2021. Introducing Formal Methods to Students Who Hate Maths and Struggle with Programming. In *Formal Methods Teaching Workshop*. Springer, 133–145.

[37] Rustam Zhumagambetov. 2019. Teaching Formal Methods in Academia: A Systematic Literature Review. In *Formal Methods – Fun for Everybody (FMFun).*