Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 22 — March 27, 2026

Patrick Lam *version 1*

Before talking about Dafny, in Lecture 17, we talked about using it to verify one particular software system—Cedar. Then we talked about Dafny for a while. Today, we're going to talk about using Dafny to write production software in general, rather than in just one case study.

The reference for today's lecture is the following recent research paper:

> Eric Mugnier, Yuanyuan Zhou, Ranjit Jhala, and Michael Coblenz. 2025. On the Impact of Formal Verification on Software Development. Proc. ACM Program. Lang. 9, OOPSLA2, Article 403 (October 2025), 27 pages. https://doi.org/10.1145/3763181

which is also available from one of the authors' webpages at `https://ranjitjhala.github.io/static/oopsla25-formal.pdf`.

Evaluating papers. When you are deciding whether or not a paper is worth your while to read, start with the title and abstract. The title's role in your decision is obvious. The abstract is a sort of summary of the paper, provided by the authors.

Here, the title is relevant to what we have just done in this course ("formal verification" and "software development"). Continuing with the abstract: it claims that so-called "auto-active" verifiers like Dafny are "sparsely used in real-world software development" and that the paper aims to "systematically identify how auto-active verification impacts software development", especially with respect to "large-scale projects". It does so by reporting on interviews with "14 experienced Dafny users". There are some words which you probably haven't encountered previously, like "auto-active" and "grounded verification", but you have encountered Dafny. Anyway, looked worthwhile to me, so you get to hear about it.

Goals of this Lecture. The goals are not as clear as for a lecture where the goal is "be able to use Dafny to verify certain methods", so here are some goals:

- know some advantages and disadvantages of using Dafny to verify real systems today;
- identify when you are in a niche where using Dafny or similar tools could help; and,
- know best practices in actually deploying Dafny.

Context

We've just spent some weeks using Dafny to verify in the small, and we talked about a case where Amazon used Dafny to verify a larger system (before switching to Lean). This paper also cites other (isolated) cases where verifiers have been used in deployed code—cloud controllers and security modules, for instance.

Section 2 of the paper defines an *auto-active* verifier as one that is not fully interactive—Rocq and Lean, which you haven't seen, are cited as fully interactive verifiers. You have seen how Dafny aims to be automatic, but sometimes it doesn't manage to prove some things, and you have to use assertions and lemmas to help it. Thus, Dafny is an auto-active verifier.

Interactive theorem provers can be powerful, but it's rare to put them in front of undergrads. Proving things with them requires expertise. Instead, we have a number of reports of students like you being able to use Dafny for classroom exercises. Why doesn't this generalize to larger codebases? Automation failure. We'll see more about that, and what people do to work around this.

Summary of Results

The paper's introduction provides a longer-form summary of its results. In bullet form:

- there are two types of developers: *formal-first* and *engineering-first* developers; each type of developer uses verification in their own characteristic way—you are likely to be engineering-first developers;
- in the design and implementation phases: verification adds specification-writing to the design phase, and specification-proving to the implementation phase. Mixed in with implementation is also debugging of failed proof attempts, as well as hardening the code so that it will verify in broader contexts than the developer's machine.
- in the testing and deployment phases: verification affects testing and code reviews, though it makes reviewing harder.

More on this below.

Methodology

For the purpose of this lecture, we should mention that there were 14 participants, who were interviewed by the paper's authors, using a semi-structured format: there was a predefined set of questions with improvised follow-ups as appropriate. The participants include (graduate) students, computer scientists, software engineers, and professors, with experience ranging from < 1 year through ≥ 10 years. The paper doesn't specify what the years of experience are in (programming in general, using formal methods, or something else). The participants worked on projects from less than 1000 lines of code to more than 10,000 lines of code (still modest by industry scale).

The full list of application domains, as provided in the paper, is: distributed systems, compilers, mathematical proof, security critical, cloud, authorization, business critical, file system, storage system, cryptography, blockchain. Certainly one can imagine how these domains might especially be interested in correct software, and amenable to specification.

Impact of Verification on Development

As we mentioned before, verification affects building software in four ways: *specification* now needs formal specifications; *implementation* also needs proofs; *debugging* includes proof debugging; and *hardening* helps proofs continue to go through even if things change.

Specification

Dafny requires specifications. How do these specifications get written?

When to specify. Different participants reported different approaches: some start with the specification; some start with an implementation, and test it on real data; and some write the specification and implementation concurrently and iterate.

What to specify. We talked about the oracle problem earlier: how do you know if an output is correct or not? Maybe you decide based on the specification. Here we are writing the specification, so where does it come from? The reported sources are: documentation (including design documents), tests, and code.

If there is documentation, then it's an obvious source for specifications. But, as the paper points out, "documentation is often incomplete and vague". Participants reported going to an informal specification first and then going from there to Dafny contracts. There's also talk of embedding textual specifications into the code using Amazon tool Duvet.

If there are tests, then they can be generalized into specifications, though it is a challenge to extract the generalization from the specific cases of tests.

The code, if it exists, should embody the specification. But it's not obvious to extract the specification from the code; it takes judgment and code-reading skills. In any case, there are a lot of implementation details that don't need to go into the specification.

Specification styles. We have talked about Dafny `method`s versus `function`s. `function`-based specifications are easier to verify: there's no need to track what has changed in the program state, aliasing, or complex control-flow. Functional specifications came up in 5 of the interviews. From the previous lecture, we had method `swap()`, which is most definitely imperative: it swaps two positions in the array, where the array is part of the program state—that is, it performs an in-place modification of the state, and (frame condition) guarantees that nothing else changes. A functional version can take an array as input and return a modified array as output, guaranteeing that nothing changes. Of course, if you are verifying a method that changes the program state, the specification can't be fully functional. It must be at least somewhat imperative, though it may isolate the changes to one step.

I find this example from Figure 3 of the paper a bit problematic, especially in terms of performance impacts (I'm not convinced these exist), but let's look at it.

```
method UpperBoundedAdditionImpl(x: uint64, y: uint64, u: uint64) returns (sum: uint64)
  ensures sum as int = UpperBoundedAddition(x as int, y as int, UpperBoundFinite(u as int));
{
```

```
  if y ≥ u {
    sum := u;
  } else if x ≥ u − y {
    sum := u;
  } else {
    sum := x + y;
  }
}

function UpperBoundedAddition(x: int, y: int, u: UpperBound): int
{
  // LeqUpperBound checks if a given integer x
  // is less than or equal to the specified upper bound u.
  if LeqUpperBound(x + y, u) then x + y else u.n
}
```

It is certainly the case that the method has three writes to local variable `sum`, and with this formulation of the method, it's hard to talk about a specific intermediate value of the variable. The functional version has no intermediate state. For arrays, in-place mutation is more efficient than returning a new array, but that's not the example we're looking at here.

There is discussion in the paper about unrestrained aliasing being a problem for Dafny verification. The problem is that you don't know that your object doesn't get changed through some alias unknown to you. I'll mention that Rust solves this problem with uniqueness. C++ also has uniqueness, but it doesn't enforce declared uniqueness.

Finally, a participant advocated that the specification should use logical types (sequences and maps, as we've seen) rather than implementation types (vectors, hash maps). What we haven't seen in class is a reported layer that "hides the details of the algorithm and exposes an interface". (There is a saying about how any problem in computer science can be solved with a layer of indirection.) This participant reports using a Dafny `predicate` to summarize what they want to prove.

Writing proofs

The whole point of using Dafny is the ability to write a proof. Out of the 14 participants, 11 of them would only be satisfied by a proof (even though it's possible that the proof might be proving the wrong thing—misled by a wrong specification). The other 3 participants sometimes trusted that their implementations are correct; two participants sometimes didn't write contracts and skipped "well-tested or relatively straight forward functions". (Another tangent: this is similar to Rust and unsafe code. Safe Rust guarantees no undefined behaviour, but unsafe Rust is allowed e.g. for performance and interoperability reasons. Unsafe Rust usually requires a textual justification of correctness, but that can be wrong.)

You should, by this point, know that it's not always easy to write Dafny proofs—and that's for the sanitized exercises that we've been doing, rather than using Dafny in anger. There is a reference to a result showing that proofs can be 10× the lines of implementations.

Some participants only committed verified code to their repository, while others committed proofs that were developed gradually, with assumptions in the place of proofs.

Regardless of when the proofs got committed, all participants developed the proofs incrementally with assumptions. Assumptions also can hide the fact that something wasn't actually proven, which is something to pay attention to.

I don't want to discuss top-down versus bottom-up, which just reminds me of discussions about how to best write code. I think it's still too early to tell.

Debugging proofs

When debugging code, you have to reproduce the problem and then figure out how to fix it. For a Dafny proof, it's more like a "prove or disprove" math problem. You're trying to prove something, but you don't even know that it's actually true: there might be an error in the code, for instance. It's useful to test the Dafny code in that case—if you can find a witness to a bug, you know that you should stop trying to prove the incorrect code.

For our purposes, we'll say that a bug means that the specification doesn't match the implementation. Participants reported this as happening often.

Participants also reported Dafny failing to prove statements without hints. Only one participant reported Dafny being wrong due to unsoundness.

We have seen Dafny failing to prove statements in the past few weeks, so I won't go on about that. But, I will summarize participant complaints and workarounds. (1) Dafny error messages don't say why Dafny failed—it should provide more information. (2) As we've discussed, participants (all of them) used assertions to figure out what Dafny and its SMT solver knows. There's an analogy to breakpoints, and a methodical approach of "asserting the post conditions at every exit point". (3) Participants also used `assume`s and conditional branches to figure out what Dafny is failing to prove.

Taking a step back, the wider challenge to using Dafny is being able to "build a mental model of implicit verifier state". When Dafny fails, you have to figure out what it "knows" at any point in the proof, to figure out why it failed. Some participants pointed out that interactive proof verifiers make explicit proof state visible, unlike Dafny. We've said that annotations can help with this. But sometimes there are too many annotations needed to encode the entire explicit state.

Proof hardening

This is somewhat analogous to code refactoring to eliminate smells. The proof works right now, but it might not work on a different machine, a different solver version, or after a small change to the implementation, as reported by 4 participants. A participant said that they made "some little fiddly change to the code, just maybe pass one more element in the state [...] and all the proofs would just stop working." Participants did not like this, describing it as "soul crushing" and inducing "existential dread". I guess you feel like the carpet got pulled out from under you. That sort of thing should not change!

Three of the participants had not encountered brittle proofs. Some addressed brittleness reactively, e.g. by increasing resource limits after hitting a brittleness problem. Others addressed brittleness proactively; one reported a month-long major refactoring, and another stopped to fix timeouts before continuing.

Why brittleness? Dafny doesn't know what's important and passes too much context to the server, though sometimes the wrong context. We talked about making state visible, but if it's gigabytes

of state, you're not going to be able to read that line-by-line. Frame conditions—stating that state doesn't change—also get large, contributing to the context.

Dealing with brittleness. Many participants monitored the solver's resource count to see if things were getting hard. Another tactic was to add (or remove) hint assertions, which can greatly reduce solving time. "Unsat core" and `assert false` can help finding assertions to add or remove. Breaking up proofs into smaller lemmas and using `opaque` and `reveal` (which we didn't talk about) were also tactics that participants used.

More systematically, 4 participants used structured approaches, including style guides, with the goal of reducing automation. In this course, using automation is fine, but it can be tricky when trying to work on larger projects.

Participants reported that Dafny is getting better on the brittleness front. Brittleness doesn't matter if the tool is not brittle on the examples you care about.

Impact of Verifiation on Deployment

We are indeed going to talk about software quality assurance and maintenance in this course. We are also going to touch on deployment of verified software.

Testing. (This *is* the testing course, after all.) At first, I was surprised to learn that Dafny had support for test cases, but it makes sense when you think about it. We've seen specification errors, after all. All participants used testing, with about half of them using it to compare the specification to the implementation or to expected behaviour.

Specifically: (1) *verified specifications*: some participants wanted an executable model of the code; the model conformed to desired properties, but wasn't what was actually deployed. Instead, there were comparisons between the model and the deployed code. (2) *verified implementations*: participants verified the code that would actually be deployed, and used testing to make sure that the deployed code behaved just like the unverified legacy code. Both (1) and (2) could use differential random testing as seen for Cedar. Finally, in many cases, (3) participants verified new implementations of new specifications; there was no legacy.

Most testing frameworks, including Dafny's, are designed for unit tests. But participants reported using both unit testing and end-to-end testing. It's unclear what infrastructure they used for end-to-end testing. One participant ran the verified implementation in parallel with the production code, and found differences, even after fuzzing. (Testing in prod!) Another participant said that people wanted to see the verified code working on real data.

There's something about testing library functions. I don't quite understand, but I think it means that the participants checked to see if the libraries did what their specifications promised.

I haven't mentioned that Dafny code can be transpiled to other languages for execution. It can be, and participants tested the resulting code for performance. There was also one report of a transpiler bug. Participants also transpiled tests and ran them to ensure that the transpiler didn't break anything.

Code Review. Code review is a best practice and all of the multi-author projects here used it. Dafny affects code review, in particular by making reviews larger—they now include specifications and transpiled code in addition to just the normal implementation. Indeed, they can end up being "quite large" compared to best practices. And the proof changes are intertwined with the code.

The participants reported focussing on the specifications rather than the "proof code" (implementations I think). When there is non-machine-verified code, e.g. for library methods, that code might specifically get reviewed.

We talked about proof hardening. One participant reported reading the proof to ensure clarity and maintainability.

Sometimes the participants looked at the transpiled code, though definitely not all the time. Lack of trust in the transpiler was one reason. Ensuring transpiling code quality was another reason, especially when other teams were likely to interact with that code.

I am skipping the discussion of packaging.

Impact of Verification on Maintenance

We have talked about how testing lets you confidently refactor and optimize your code. Verification is even better! You can likely rely on your verified specification more than you can rely on your test suite. In particular, Dafny will tell you if your implementation no longer matches the specification. Tracking breaking changes in your libraries is generally a hard problem.

It is true that a change will often require modifications to both the specification and the proof, so that's more work. But you get more knowledge about your system out of it: you know that something might or might not break as a result of the change.

When there is a breaking change, then you can see which specification no longer holds, and focus testing accordingly. This lets you use small fixes for the specification and the proof. And, again, you also get some confidence that a particular change is not breaking if Dafny agrees that it still respects the same specification.

All this relies on having an initial deployment of verified code, so that you can evaluate differences against that deployment.

On the topic of optimization: participants reported being willing to make changes that they wouldn't otherwise be, for instance: localized changes that do not affect the specification; algorithmic optimization; and removing redundant checks. Verification shows that the new code continues to meet the specification. Redundant checks include those that can be shown at proof time with `requires` clauses.

Summary

We talked about how people do indeed use Dafny on real systems, perhaps via transpilers. This affects all stages of software development, but gives software that meets its formally-expressed specifications, and enables lower-risk changes.