Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 17 — March 9, 2026

Patrick Lam *version 1*

Today, we are going to start talking about Dafny, which is a tool for automatically verifying annotations in program code. As I've alluded to before, Dafny makes it possible to mostly-automatically do what you have done manually in SE 212. Some people call it an "auto-active" system (versus interactive theorem provers).

Dafny Application: Cedar authorization-policy language

I think some people are using Dafny "for real". Here's an example I managed to find, about the Cedar authorization-policy language[1] at Amazon. It looks like the formalization was later migrated from Dafny's automatic verification to another system, Lean, which supports interactive theorem proving (not as automatic, but more powerful), and there is a description of Cedar and Lean in [**?**].

Here is the blog post about Dafny and Cedar.

`https://www.amazon.science/blog/how-we-built-cedar-with-automated-reasoning-and-differential-testing`

What Cedar does. A Cedar user writes policies in a domain-specific language and then can execute queries on the policies, which may have response "allow" or "deny". Consider a hypothetical todo list manager TinyTodo. It may have the following policies, written in Cedar's language:

```
// policy 1
permit(principal, action, resource)
when {
  resource has owner && resource.owner == principal
};
```

From the blog post: "This policy states that any principal (a TinyTodo **User**) can perform any action on any resource (a TinyTodo **List**) as long as the resource's creator, defined by its **owner** attribute, matches the requesting principal." This above policy that enables access in some cases. Here's one below that forbids access.

```
// policy 3
forbid (
 principal in Team::"interns",
 action == Action::"CreateList",
 resource == Application::"TinyTodo"
);
```

[1] `https://www.cedarpolicy.com/en`

What we have here is that interns can't create new lists for application TinyTodo.

Given a complete set of policies, then TinyTodo doesn't need to do the reasoning about authorization; it can just ask Cedar whether a given action is to be allowed or denied, and proceed accordingly.

Verification-guided development. The point of Dafny (and Lean) is to be able to prove things about software. We write some properties and some code and formally verify that the code respects the properties. It's better than testing, in that the property is guaranteed to be true of the code.

Maybe we don't actually want to deploy Dafny code in production. (Actually, it is possible to compile Dafny to other languages and deploy that). In this application, the Dafny code we're talking about is best thought of as a model of code written in some other language (in this case, Rust). It is still useful to verify the properties on the Dafny implementation, because we'll know that it is possible to write code respecting them. Plus, the Dafny code can serve as an oracle, which is useful, and we can look for divergences between the Dafny code and the Rust code.

But, really, we want to know that the Rust code satisfies the properties. Safe Rust is going to ensure that there is no undefined behaviour, so that's useful, but you do not get any guarantees that the behaviour respects the contracts.

You can put in assertions, but what do you do if they fail?

The approach here is to use *differential random testing* to make sure that the Rust implementation matches the Dafny model. This is like fuzzing: generate millions of inputs and check that Rust and Dafny do the same thing.

Properties of Cedar. There are two overarching properties that Cedar must satisfy when it responds to queries. Quoting the post again:

- *explicit permit*: permission is granted only by individual **permit** policies and is not gained by error or default;
- *forbid overrides permit*: any applicable **forbid** policy always denies access, even if there is a **permit** policy that allows it.

So, authorization has to be explicitly granted by some specific policy, and if there is any forbid policy that affects a request, then access is forbidden.

For instance, a request might be *principal*=**User::"Alice"**, *action*=**Action::"GetList"**, and *resource*=**List::"AliceList"**. Applying that to policy 1, we get expression **List::"AliceList" has owner && List::"AliceList".owner == User::"Alice"**. This is probably going to be true, in which case the request *satisfies* the policy. More generally, the authorization engine computes the request over all of its policies and returns an overall decision.

Some Bad Code. OK, let's look at some buggy Dafny code that tries to compute whether a request is authorized.

```
function method isAuthorized(): Response { // BUGGY VERSION
  var f := forbids();
```

```
  var p := permits();
  if f ≠ {} then
    Response(Deny, f)
  else
    Response(Allow, p)
}
```

Can you see why this code is wrong? Well, it does correctly capture forbidding a request if there is some reason to forbid it. But we also want a policy that says that there has to be an explicit permit for a request to be allowed, and this code doesn't respect such a policy. We can encode that policy as a predicate (which uses a lemma).

```
// A request is explicitly permitted when a permit policy is satisfied
predicate IsExplicitlyPermitted(request: Request, store: Store) {
  ∃ p •
    p in store.policies.policies.Keys ∧
    store.policies.policies[p].effect = Permit ∧
    Authorizer(request, store).satisfied(p)
}

lemma AllowedIfExplicitlyPermitted(request: Request, store: Store)
ensures // A request is allowed if it is explicitly permitted
        (Authorizer(request, store).isAuthorized().decision = Allow) ⟹
        IsExplicitlyPermitted(request, store)
{ ... }
```

The predicate is saying that there has to be a specific policy `p` which permits `request`. The lemma is something that needs to be shown true; here, it is saying that if the authorizer says yes, then the predicate `IsExplicitlyPermitted` has to be true. When Dafny verifies the function `isAuthorized`, it tries to check that the lemma holds. But it doesn't, and supposedly Dafny reports "A postcondition might not hold on this return path" pointing at the ensures clause.

```
// BUGGY VERSION
function method isAuthorized():
    Response {
var f := forbids();
var p := permits();
if f ≠ {} then
  Response(Deny, f)
else
  Response(Allow, p)
}
```

```
// FIXED VERSION
function method isAuthorized():
    Response {
  var f := forbids();
  var p := permits();
  if f = {} ∧ p ≠ {} then
    Response(Allow, p)
  else
    Response(Deny, f)
}
```

Dafny verifies the fixed version, and also the mirror property *forbid overrides permit*, which we don't show.

Dafny also can show other properties, including that the Cedar *validator* is *sound*: as they put it, "if the validator accepts a policy, evaluating the policy should never result in certain classes of error". Making this proof go through in Dafny enabled the developers to fix subtle bugs in the design, which would otherwise have carried over into the implementation.

The specification is useful for automated reasoning by Dafny, but also for manual reasoning by people. Even good Rust is still an implementation language with lots of implementation details. Reading the Dafny (or Lean) is much easier, since it has much less code (1/6th at the time of report).

Dafny and Rust. OK, so we can formally verify the Dafny. The thing about models is that I have always worried about making sure that the actual implementation conforms to the model. If you have no proof of correspondence, then, sure, the model can help, but bugs can sneak into the implementation as well. The solution here is *differential random testing*.

The Amazon people used techniques that we've talked about in this class to overcome this problem. They randomly generated millions of inputs (access requests, data, and policies) and check that Dafny (now Lean) and Rust produce the same output. If they disagree, there is a bug.

The inputs aren't completely randomly generated. They used the same techniques that we've talked about in this class—they use input generators to create "policies, data and requests that are consistent with one another". They report 6 hours nightly of DRT and 100 million total tests. They found errors in their libraries (IP address operations and parsing, as well as the policy parser).