Runtime Tests for Dafny

Authors

Institutes

Abstract. Dafny is a verification-aware programming language that can automatically prove that the implementation of a particular function satisfies its specification. Dafny provides compilers for generating executable code in different target languages, such as C# or Java. A target-language executable generated from Dafny source must, in theory, exhibit the runtime behavior proven by Dafny verifier. In practice, however, compiler bugs and incorrect specification of external functions can lead to the executable producing a result that is different from the one expected. To help ensure that the verified behavior carries over to the target language, we propose to introduce runtime tests to Dafny. Supported features include mocking and parameterized tests.

Keywords: Dafny · Testing

1 Motivation

An effective testing framework supports several strategies and levels of testing, which makes it versatile enough to be used at different stages of the development process. We built our testing framework around the need to support all of unit-, integration-, and system-level testing, with both automatic and manual test generation in mind.

In *unit-level testing*, the developer is only concerned with the correctness of a single function - all the outgoing function calls can be stubbed to mimic a particular scenario. We introduce this kind of testing to Dafny by supporting the automatic compilation of external methods that return mocked objects. The compiler uses the postconditions constraining the returned object in Dafny to set up the mock in the target language using the appropriate mocking library.

In *integration-level testing*, the developer is usually concerned with how several objects of different types interact with each other. The way these objects are constructed is irrelevant - only their internal state at the moment of interaction is. We introduce this kind of testing to Dafny by supporting the automatic compilation of external methods that return fresh objects of a requested type. This allows the developer to bypass the process of choosing an appropriate constructor and constructor arguments for initializing an object. Inferring appropriate constructor arguments is not a trivial task and the ability to bypass it is, therefore, also crucial for automatically-generated tests.

In *system-level testing*, the developer typically provides a sequence of API calls to ensure that they produce some desired result at runtime. This kind of

testing does not require the addition of any special features to the Dafny language aside from those features that are shared by all runtime tests, as described below.

All runtime tests for Dafny make use of the *assertion library*, which is designed so that the behavior tested at runtime is also verified by Dafny before compilation.

Ideally, all semantic behaviors present in the Dafny model are preserved after cross-compilation to C# or Java. However, we have found this is not the case. By including additional well-known assertion libraries for our target languages, we can discover such bugs in the cross-compiler.

Figure 1 details an example of impedance mismatch. Due to the type constructor, Dafny will not allow the assignment of a negative integer to an uint32-typed variable. However, Java has no primitive type for unsigned integers, so the constant is converted by the Java compiler to a signed integer.

NativeTypes.dfy

```
module NativeTypes {
    newtype uint32 = i:int | 0 ≤ i < 0x100000000
    const UINT32_MAX :uint32 := 0xffffffff;
}
```

NativeTypes.java

```
public static int UINT32__MAX()
{
    return -1;
}
```

Fig. 1: An example of a Java compiler bug. UINT32__MAX is defined as -1 despite its type definition.

2 Design and Implementation

Support for mocking and unit testing is, implementation-wise, the most complex part of the proposed testing framework. The challenge lies in mapping the post-conditions written in Dafny to method calls in the target language's mocking framework. Table 1 illustrates how this mapping is carried out in C# and Java. We support the most general approach to function call stubbing that is also supported by the target mocking frameworks - the user can supply an arbitrary expression over the function arguments that is evaluated instead of the function body at runtime. Additionally, the developer can specify stubbed function calls as implications, where the premise determines when a function call should be stubbed and the consequence describes how it should be stubbed.

Despite this generality, not any Dafny postcondition can be successfully compiled in this manner - this is because such a postcondition may not always equate

the result of a given function call to a specific value and may instead constrain the result in terms of other function calls. For instance, a postcondition might define a boolean formula over several function predicates. Solving for exact values that the predicates must evaluate to is, in this case, an instance of the satisfiability problem, which is NP-complete. Hence, we are only able to compile a select portion of Dafny postconditions that can be mapped directly to method calls in the target mocking frameworks. Figure 2 provides the grammar of supported postconditions.

```
S        = FORALL
         | EQUALS
         | S && S
EQUALS   = ID.ID (ARGLIST)  == EXP // stubs a method call
         | ID.ID            == EXP // stubs field access
         | EQUALS && EQUALS
FORALL   = forall SKOLEMS :: EXP ==> EQUALS
ARGLIST = ID  // this can be one of the skolem variables
         | EXP // this exp may not reference any of the skolems
         | ARG_LIST, ARG_LIST
SKOLEMS = ID : ID
         | SKOLEMS, SKOLEMS
```

Fig. 2: The grammar of postconditions supported by the mocking framework. Non-terminal `EXP` stands for an arbitrary Dafny expression, `ID` stands for a variable or type identifier.

Because methods, as opposed to functions, may have side-effects, Dafny does not admit method calls in postconditions. This makes stubbing methods that have post-conditions impossible. In particular, we do not allow stubbing methods in such a way that they return different values after a certain number of calls.

Another difficulty in connecting the source and target framework stems from Dafny's use of proprietary types. This presents issues when writing parameterized unit tests with a method source, as the return type differs from what the testing libraries expect. Our solution injects an intermediary method source that handles the conversion.

Our implementation requires each parameterized test and method source to be a Dafny method. The method source must be declared static. The return type of the method source must be a sequence of tuples. Lastly, the types inside each tuple must match the types of the test method's parameters.

3 Conclusions and Future Work

The proposed testing framework opens new possibilitie for automatic test generation. Since the expected runtime behavior of a function is specified by its

Meaning	**Dafny**	**Java (JUnit/Mockito)**	**C# (Xunit/Moq)**
A test method with no arguments.	`{:test}` attribute	`@Test` annotation	`[Fact]` annotation
A test method executed multiple times with arguments supplied by the provider method `M`.	`{:test M}` attribute	`@ParameterizedTest` and `@MethodSource` annotations	`[Theory]` and `[MemberData]` annotations
Method returns an unconstrained fresh object	`{:fresh}` attribute	Exploiting the zero-argument constructor	
Method returns a mocked object of type `Clazz`.	`{:mock}` attribute	`Mockito .mock(Clazz.class)`	`new Moq.Mock<Clazz>() .Object`
Stubbing function `Do` of the mocked return parameter	`ensures o.Do() == 0`	`Mockito .doReturn(0) .when(o).Do()`	`m.Setup(o=>o.Do()) .Returns(0)`
Stubbing a function call under specified conditions	`ensures forall a,b:int :: a == b ==> o.Do(a, b) == 0`	Custom Argument Matcher for converting a predicate of n arguments into n predicates of one argument each.	
Stubbing a function call with a return value that depends on the arguments	`ensures forall a:int :: (o.Do(a) == a)`	Passing a lambda to Mockito's `thenAnswer`	Passing a lambda to Moq's `Returns`

Table 1: The mapping from introduced method annotations and supported post-conditions in Dafny to the corresponding instructions in Java and C#.

postconditions, it should be possible to automatically generate tests that verify that the expected behavior is observed at runtime. Such a testing framework would be invaluable for testing new compilers and would provide additional level of assurance to developers who use a lot of external functions.