# Delfy: Dynamic Test Generation for Dafny

Master's Thesis Project Description

Patrick Spettel
Supervisor: Maria Christakis

June 2013

## 1 Introduction

Dafny is a language and program verifier for functional correctness which has been developed by Microsoft. The language is designed to support static program verification. The idea is to annotate the program using special verification features such that functional verification can be performed automatically [1]. The verifier is sound and powered by Boogie [2, 3] and Z3 [4].

It is possible that in certain circumstances the verifier is not able to prove or disprove correctness. Reasons for this include that the verifier has not enough information or it times out. Thus, the idea of this project is to build a dynamic test generation tool for Dafny which can be used to complement the static verifier.

Dynamic test generation combines concrete and symbolic execution [5]. The unit in test is executed for arbitrary concrete input values. This concrete execution aids the symbolic execution whenever it is stuck. The symbolic execution builds up a path constraint using symbolic values to represent the input values. This path constraint is then modified (e. g. parts are negated) and passed to a solver to find inputs which exercise new paths [6–8]. This idea can be used to test any assertion in the code.

## 2 Project Definition

This section specifies the goals of the project. It is split into a core and an extension part. The focus first lies on the core part. The extensions will then be implemented building on the results of the core part.

## 2.1 Core

The main goal of the core part of this project is to build a dynamic test generation tool for the compilable subset of the Dafny language (classes, methods, function methods, algebraic datatypes and the compilable statements and expressions). Algorithm 1 shows in pseudo code the main idea of dynamic test case generation. The tool will be implemented such that it is as stable and as configurable as possible, i. e. the search strategies will be exchangeable easily, both whole-program testing and unit-testing will be supported in a configurable way and the communication with Z3 will be extendable. This means that it will be easy to support parallelized queries and use Z3's learning strategies.

**Algorithm 1** Pseudo code of dynamic test generation[1]

```
procedure EXPLORE(Seq<Condition> prefix)
    values := solve(prefix);
    if solution is available then
        path := executeConcrete(path);
        pathCondition := executeSymbolic(path);
        extension := pathCondition[|prefix|...];
        for all non-empty prefixes p of extension do
            if p = p' ++ [branchCondition(c)] for some p',c then
                explore(prefix ++ [p'] ++ [branchCondition($\neg c$)]
            end if
        end for
    end if
end procedure
```

The first step is to build the symbolic execution engine for a relatively easy subset of the compilable features of the language. This means that, in a first version, the tool will only support integers and it will only allow dynamic test generation for methods. After this more language features will be added.

Dafny supports many specification constructs which are not compiled to executable code. The goal of the core part is to support all the language features which compile. If specification constructs are needed in order to perform dynamic test generation, the Dafny compiler will be extended to support these.

[1] Algorithm taken from slides of the spring semester 2012 ETHZ course "Software Architecture and Engineering" by Prof. Peter Müller.

## 2.2 Extensions

If we are able to fully verify a Dafny program there is in principle no need for dynamic test generation. But there are many cases in which we cannot achieve full functional correctness of a program. Either annotations are missing or the verifier is not able to perform the proof automatically. So there can be a huge gap. On one extreme there are programs with essentially no specification annotations. On the other extreme there are programs which are annotated in a way such that full functional correctness can be proven automatically. The extension part of the project aims to bridge this gap to some extent.

The main idea is to combine verification and testing. Perhaps there are cases in which some parts of the program can be proven but others not. We then want to take advantage of the facts which are proven.

As an example assume that there is a function with pre- and postconditions and symbolic execution is not feasible for this function. This can be the case if there can be infinitely many paths in this function. An example of this is if the function contains a loop with a condition which depends on parameters to the function. Moreover, assume that the verifier was able to prove the loop invariant and $pre \implies post$. Thus, we aim to take advantage of the loop invariant or $pre \implies post$ to generate useful test cases even if we cannot symbolically execute the function. An advantage of this is that we do not generate tests for verified code which yields a smaller test suite.

The other way is also possible. If annotations are missing or cannot be proven we want to use dynamic test generation to build up confidence that the code and its specification are correct.

This is what the extensions are about. We will extend the core part with support for the following features:

**Pre- and postconditions:** We will make use of verified pre- and postconditions in such a way that we do not need to symbolically execute these methods or function methods.

**Modifies clauses** will be used to simplify updating the symbolic state during symbolic execution of methods.

**Loop invariants:** In this extension we will use loop invariants to deal with the problem of the possible non-termination when symbolically executing loops.

**Non-determinism:** In Dafny there are constructs which are non-deterministic such as the non-deterministic if-statement which is shown in Listing 1. We will support these by introducing special conditions into the path constraint. That is we need to treat all symbolic values as usual up to the non-deterministic statement. From then on there are multiple possible execution paths which we must take into account during symbolic execution.

Listing 1: Non-deterministic if-statement

```
        if (*) {
          ...
        }
```

**Stratified inlining** is an idea to deal with the possible non-termination of symbolic execution of loops. Instead of symbolically executing one loop at a time until the loop iteration bound is reached, we will keep track of all loops during symbolic execution and execute every loop one time, then two times, then three times, and so on.

By taking advantage of these features for dynamic test generation we are then going to evaluate whether we can do better than other tools of this area.

The above list is not limiting or exhaustive of the features we will target.

## 3 Project Management

This section describes the coarse time management of the project.

### 3.1 Overall Project Plan

**Project Start:** June 1st 2013

**Project End:** December 1st 2013

The project is split into four phases, the *Getting Started Phase*, the *Design and Implementation Phase*, the *Evaluation Phase* and the *Writing Phase*. These phases can possibly overlap. I especially intend to already collect notes and do some write-up during the first phases. I think this is a good idea because it helps to document the whole work which is done throughout the project. Moreover, these notes then also help to write the final project report. Evaluation is also not a single phase but we already evaluate the tool on an ongoing basis to some extent during the implementation phase.

### 3.2 Getting Started Phase

The *Getting Started Phase* phase is meant for getting familiar with the environment. In particular it includes setting up all the required tools, getting familiar with the tools and reading about all the details needed to get started. This includes getting familiar with the source code and understanding how to communicate with Z3.

Maria provided me with pointers to relevant papers and tools which I was able to study already during the spring semester 2013 to some extent. So I already start to implement the symbolic execution engine for the core part.

**Projected time frame:** first half of June 2013

**Deliverables:**

- Initial presentation to show what has been achieved so far

### 3.3 Design and Implementation Phase

In this phase the design for the tool is worked out and the tool is implemented. As described in section 2, the implementation is split into core features and extension parts. The focus first lies on the core features. Based on this core the extensions are then being implemented.

**Projected time frame:** mid of June 2013 to mid of October 2013

**Deliverables:**

- Design documents
- Documentation
- Source code
- Intermediate presentation

### 3.4 Evaluation Phase

In the evaluation phase experiments are done using relevant test sets. The goal is to compare the tool to other tools in this area, e. g. Pex [9]. We then evaluate whether we can do better if we are able to take verification constructs into account. And if yes, in which cases. Moreover, it is also important to get to know possible weaknesses of our tool.

**Projected time frame:** mid of October 2013 to mid of November 2013

**Deliverables:**

- Experimentation results
- Evaluation/Conclusions

### 3.5 Writing Phase

The writing phase is used to document the project work. I intend to gather material throughout the previous phases. This phase is then used to put all things together nicely.

**Projected time frame:** second half of November 2013

**Deliverables:**

- Final report
- Final presentation

### 3.6 Meetings

Meetings are held on a regular basis (e. g. every week). If problems arise meetings are also held whenever needed. In a meeting, problems and the status relative to the previous meeting are discussed. Moreover, the steps to be done next are discussed.

## References

[1] K. Rustan M. Leino, "Dafny: An automatic program verifier for functional correctness," in *LPAR*. 2010, vol. 6355 of *LNCS*, pp. 348–370, Springer.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *FMCO*. 2006, LNCS, p. 364–387, Springer.

[3] K. Rustan M. Leino, "This is Boogie 2," June 2008.

[4] Leonardo De Moura and Nikolaj Bjørner, "Z3: An efficient SMT solver," in *TACAS*. 2008, vol. 4963 of *LNCS*, pp. 337–340, Springer.

[5] James C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, July 1976.

[6] Koushik Sen, Darko Marinov, and Gul Agha, "CUTE: A concolic unit testing engine for C," in *ESEC*. 2005, pp. 263–272, ACM.

[7] Patrice Godefroid, Michael Y. Levin, and David Molnar, "Automated whitebox fuzz testing," in *NDSS*. 2008, The Internet Society.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen, "DART: Directed automated random testing," in *PLDI*. 2005, pp. 213–223, ACM.

[9] Nikolai Tillmann and Jonathan de Halleux, "Pex—White box test generation for .NET," in *TAP*. 2008, vol. 4966 of *LNCS*, pp. 134–153, Springer.