Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Delfy: Dynamic Test Generation for Dafny

Master Thesis Report

Patrick Spettel

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

Zurich, June - November 2013

Supervised by: Maria Christakis
Prof. Dr. Peter Müller

Abstract

In this Master's thesis we present the design and implementation of a novel dynamic test generation tool. The novelty is that it targets Dafny, a language designed with static verification in mind. The core is based on existing ideas in dynamic test generation but special verification constructs pose challenges. We describe the design and implementation and how we have approached the challenges. We then present our results to show what we have achieved and learned throughout the whole process of this project.

Acknowledgments

I would like to thank Maria Christakis for her supervision of this project. I am very thankful for the guidance through the project and for the various helpful discussions on the different challenges I faced.

I would also like to thank Prof. Dr. Peter Müller for providing me the opportunity to do my Master's thesis project at the Chair of Programming Methodology. Thanks also to all the members of the Chair of Programming Methodology. The feedback and ideas which were provided during my presentations were very helpful. Special thanks to Malte Schwerhoff, Valentin Wüstholz and Leonardo de Moura for supporting me in challenges I had with Dafny and Z3.

This thesis is dedicated to my parents for all their support.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

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 [12]. The verifier is sound and powered by Boogie [1, 11] and Z3 [4].

However, 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, has too much information or it times out. For such cases it is helpful to use software testing methods. But manually finding inputs that detect errors is a difficult task.

Thus, the goal of this Master's thesis project is to build a dynamic test generation tool for Dafny, which can be used to complement the static verifier. The motivation for the usage of the tool is two-fold. Firstly, errors can be found if the Dafny static verifier is not able to automatically reason about a program. And secondly, we can achieve full coverage in some cases with the usage of dynamic test generation. This case is especially interesting if Dafny has not enough information available to prove a particular statement.

1.2 Background

1.2.1 Dafny

Dafny is a feature-rich verification language. Therefore, we do not give an introduction here. An overview of Dafny is given in [12]. Moreover, the project web page ([13]) gives further pointers. For a tutorial-like introduction, [14] is especially interesting because Dafny is provided in the browser side-by-side with the tutorial.

1.2.2 Concrete execution

In a concrete execution, inputs to a program are fixed. Then the program is executed using these inputs. Different approaches exist for keeping track of the path taken through the program. One such approach is to create a trace file that can then be analyzed by other tools. Another idea is to combine other tools directly with the concrete execution, for example through callbacks.

1.2.3 Symbolic execution

Symbolic execution has been introduced by James King. The idea is to statically simulate the execution of a program using symbolic values for inputs. For this, a symbolic state is kept. This state consists of a prefix of a path through the program, a mapping of variables to expressions over symbolic variables, a symbolic heap and path conditions. A path condition is a condition over symbolic variables that holds if and only if the execution takes a certain path [9].

As an example, Listing 1.1 shows a simple Dafny program. This program consists of a single method `testme` that takes an integer x as an argument. It then prints the integers beginning with x up to but excluding 2.

Table 1.1 summarizes the updates to the symbolic state when calling this method with the integer 0. A row in this table shows the symbolic state and the path condition after executing the code on the line with the given line number. Capital letters denote the symbolic versions of the corresponding variables. The `while` statement acts as a branch which is tested in every iteration. Thus, when $x = 2$ the loop is not entered anymore which is also reflected in the path condition.

```
1  method testme(x: int)
2    requires 0 <= x;
3  {
4    var i := x;
5    while (i < 2)
6    {
7      print i;
8      i := i + 1;
9    }
10 }
```

Listing 1.1: Example to illustrate symbolic execution.

Line	Symbolic state	Path condition
1	$x \to X$	$true$
2	$x \to X$	$0 \leq X$
3	$x \to X$	$0 \leq X$
4	$x \to X, i \to X$	$0 \leq X$
5	$x \to X, i \to X$	$0 \leq X \wedge X < 2$
6	$x \to X, i \to X$	$0 \leq X \wedge X < 2$
7	$x \to X, i \to X$	$0 \leq X \wedge X < 2$
8	$x \to X, i \to (X + 1)$	$0 \leq X \wedge X < 2$
5	$x \to X, i \to (X + 1)$	$0 \leq X \wedge X < 2 \wedge (X + 1) < 2$
6	$x \to X, i \to (X + 1)$	$0 \leq X \wedge X < 2 \wedge (X + 1) < 2$
7	$x \to X, i \to (X + 1)$	$0 \leq X \wedge X < 2 \wedge (X + 1) < 2$
8	$x \to X, i \to ((X + 1) + 1)$	$0 \leq X \wedge X < 2 \wedge (X + 1) < 2$
5	$x \to X, i \to ((X + 1) + 1)$	$0 \leq X \wedge X < 2 \wedge (X + 1) < 2 \wedge \neg(((X + 1) + 1) < 2)$

Table 1.1: Summary of symbolic state updates for the example in Listing 1.1 with $x = 0$.

1.2.4 Dynamic test generation

Dynamic test generation combines concrete and symbolic execution to dynamically generate test inputs for programs. Goals include the maximization of branch coverage and the discovery of faults in a given program.

During concrete execution, the unit under test is executed for arbitrary concrete input values. At the same time, the symbolic execution builds up a path condition using symbolic values to represents the input values. This results in an expression over symbolic values which represents the path taken through the program by the concrete execution. This path condition is then modified (e. g. parts are negated) and passed to a solver to find inputs which exercise new paths [18, 7, 6].

Using the example in Listing 1.1 from Section 1.2.3 we have the path condition $0 \leq X \wedge X < 2 \wedge (X + 1) < 2 \wedge \neg(((X + 1) + 1) < 2)$. In dynamic test generation the conjuncts of the path condition are tagged with the types of branches they come from. For example the first conjunct, $0 \leq X$, comes from a precondition. This means that we do not modify it. Otherwise we would potentially break the precondition when calling the method. One example of a modification is $0 \leq X \wedge X < 2 \wedge \neg((X + 1) < 2)$ in which we leave out the last conjunct and negate the second-to-last conjunct. This yields $x = 1$ as another test input.

Chapter 2

Design of the dynamic test generation engine

Algorithm 1 illustrates the core of our dynamic test generation engine.[1] It is pseudo-code that shows how the branches of a program are explored. The procedure EXPLORE takes a prefix of a path condition as an argument. This prefix represents the condition of the path we want to follow. The expression over symbolic variables is then solved using a theorem prover. If the condition is satisfiable, this yields an assignment of the symbolic variables to concrete values. These concrete values are then used to concretely execute the unit under test. Afterwards, the symbolic execution yields a new path condition. The extension of the prefix of this path condition is modified in order to exercise new branches in the next call to EXPLORE.

Algorithm 1 Pseudo-code of dynamic test generation

```
1:  procedure EXPLORE(Seq<Condition> prefix)
2:      values := solve(prefix);
3:      if solution is available then
4:          path := executeConcrete(values);
5:          pathCondition := executeSymbolic(path);
6:          extension := pathCondition[|prefix|...];
7:          for all non-empty prefixes p of extension do
8:              if p = p' ++ [branchCondition(c)] for some p',c then
9:                  explore(prefix ++ p' ++ [branchCondition(¬c)]
10:             end if
11:         end for
12:     end if
13: end procedure
```

We took this idea and designed Delfy with configurability and extensibility in mind. Figure 2.1 shows a high-level class diagram. The entry point into Delfy is in the class `DelfyDriverMain`. The class `DelfyMain` contains the code that glues the exploration together. Everything that is underspecified in the pseudo-code is designed to be extensible and configurable in Delfy. In particular, we do not fix how the execution is done, how conditions are solved and how the next condition to explore is computed. We designed this using generalization. The exploration routine simply gets a configuration and uses the API of the abstract classes. Different specific behaviors

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

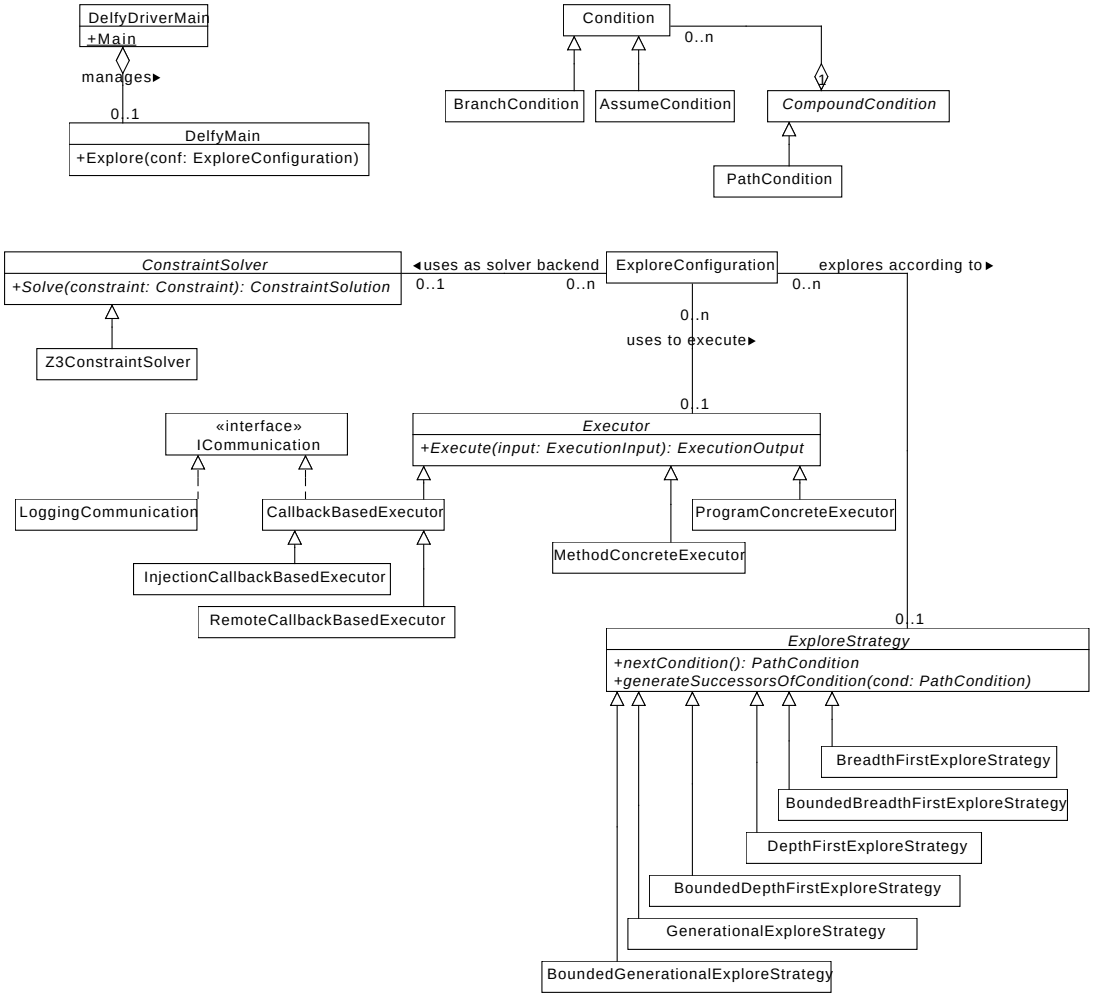

Figure 2.1: High-level Delfy class diagram.

are then implemented in subclasses.

In the next Sections we explain the design of Delfy regarding the execution of units under test, the condition solving and the exploration order, respectively.

2.1 Execution

In Algorithm 1, concrete and symbolic execution is separated. First, the unit under test is concretely executed. This results in a trace, which is then passed to the symbolic execution step. In contrast to this approach we combine these two steps into one. That is, *during* the concrete execution we update the symbolic state. This is achieved through callbacks at clearly defined points in the concrete execution. The next two Sections, Section 2.1.1 and Section 2.1.2, explain this approach in detail.

2.1.1 Concrete execution

We extended the Dafny compiler for the concrete execution of Dafny programs under test. The Dafny compiler works by translating Dafny to C#. The resulting C# code is then compiled into a .NET assembly using the C# compiler.

Our extensions are the compilation of ghost state, the generation of run time checks and the instrumentation with callbacks into Delfy.

Compilation of ghost state and run time checks: Dafny supports statements that are only used for verification and are not compiled. Assertions, pre- and postconditions and loop invariants are examples of this.

For instance, in order to be able to tell during the concrete execution whether an assertion holds or not, we need some code generated for these Dafny features. For this we use Code Contracts [16]. For example, the statement `assert false` would be translated into `Contract.Assert(false)`.

Delfy callbacks: We needed to instrument the translated code with callbacks into Delfy for our approach of doing the symbolic execution during the concrete execution.

At well-defined points in the concrete execution we call into Delfy such that we can update the symbolic state as needed. An example of such a callback is when executing an if statement. We need to call into Delfy and communicate whether the then-branch or the else-branch was taken. Moreover, the symbolic execution needs to know the expression of the if statement's guard in order to update the path condition. Section 2.1.2 explains in detail how this works in Delfy.

2.1.2 Symbolic execution

This Section explains the communication between the unit under test and Delfy, i. e., how we call back from the unit under test into Delfy. For illustration we use the if statement as an example. We do not focus on how we update the symbolic state. This is explained in detail in Sections 3 and 4.

The overall idea is to call back from the unit under test into Delfy at well-defined points. We designed this to be flexible. Referring to Figure 2.1 every class that implements the interface `ICommunication` is able to handle callbacks. The details of the communication then depend on the concrete implementation. We implemented two approaches that we refer to as the *injection*-based approach and the *remote*-based approach. These are implemented in the classes `InjectionCallbackBasedExecutor` and `RemoteCallbackBasedExecutor`, respectively.

The injection-based approach works by instrumenting the Dafny-translated C# code with a static field of type `ICommunication` to which we refer to `DafnyToDelfy`. For concrete execution we then compile this instrumented C# code into a .NET assembly and load this assembly dynamically. Using reflection we then inject the callback handler, i. e., we set the static field `DafnyToDelfy`.

The advantage of this approach is that from the unit under test we call directly back into Delfy. This has the disadvantage that we need to reference the Delfy assembly when compiling the unit under test because we make use of the interface `ICommunication` in the helper code.

The remote-based approach works by making use of remote method calls. Our implementation makes use of the Windows communication Foundation (WCF) framework [15]. Similarly to the injection-based approach we need to instrument the Dafny-translated C# code with some helper code. In this case we need to create a communication channel to Delfy. On the Delfy side the handler needs to be set up to act as a server that dispatches the callbacks.

One advantage of this approach is that the program under test can be compiled independently of Delfy. During concrete execution it is then tried to build a channel to Delfy dynamically. Another advantage is that the unit under test and Delfy can be run on different machines without changing anything. The disadvantage is the indirection through remote calls. Because during a concrete execution there can be, and usually are, lots of callbacks this can reduce performance.

In order to explain how we communicate Dafny code such as statements and expressions to Delfy we come back to the example of the if statement.

```
method testme(x: int)
{
  if (x == 10) {
    assert false;
  } else {
    print "success";
  }
}
```

Listing 2.1: Dafny example to illustrate the communication of Dafny code to Delfy.

Listing 2.1 shows a method `testme` that takes an integer x as argument and fails if x equals 10. If we concretely execute the then-branch we need to call back into Delfy saying that $x == 10$ holds, for the else-branch that $\neg(x == 10)$ holds.

Because we instrument C# code with Delfy callbacks we needed to come up with a way to embed the Dafny code we want to translate into the C# code. We came up with the following solution. During instrumentation we pretty-print the Dafny code into a string that we pass as an argument to the callback. In Delfy we can then make use of the Dafny parser and resolver to parse and resolve the expression back. For this to work we also pass all the code that is needed for resolving such as type information, functions, methods and classes. So we need to pass a full Dafny program. In particular we generate wrapper code to pass single expressions or statements.

For this example it looks like shown in Listing 2.2. Because we need a way to find the expression we wanted to communicate we use an assignment to the variable `delfyInstrumentationVar` in a new block. For the communication of statements we do not need the assignment but we also use a new block to find the statement.

```
...
if (x == 10) {
  GetDafnyToDelfy().Branch(@"
    var x: int;
    {
      ghost var delfyInstrumentationVar := (x == 10);
    }
  ");
  ...
} else {
  GetDafnyToDelfy().BranchNegated(@"
    var x: int;
    {
      ghost var delfyInstrumentationVar := (x == 10);
    }
  ");
  ...
}
...
```

Listing 2.2: C# code of the Dafny example (Listing 2.1) to illustrate the communication of Dafny code to Delfy.

The `ICommunication` interface

Listing 2.3 shows all the callbacks. They are gradually explained in Chapters 3 and 4.

```
using System;
using System.Numerics;
using System.ServiceModel;

namespace Delfy {
  namespace DafnyToDelfy {

    /// <summary>
    /// The communication interface between Dafny and Delfy.
    /// </summary>
    [ServiceContract]
    public interface ICommunication {
      [OperationContract]
      void ScopeBegin();
      [OperationContract]
      void ScopeEnd();
      [OperationContract]
      void Branch(string guard);
      [OperationContract]
      void BranchNegated(string guard);
      [OperationContract]
      void NonDeterministicWhileStmtBranch(string var);
      [OperationContract]
      void NonDeterministicWhileStmtBranchNegated(string var);
      [OperationContract]
      void Statement(string stmt);
      [OperationContract]
      void FunctionCallBefore(string functionCallExpr);
      [OperationContract]
      void FunctionCallAfter(string functionCallExpr);
      [OperationContract]
      void CallStmtBefore(string callStmt);
      [OperationContract]
      void CallStmtAfter(string callStmt);
      [OperationContract]
      void LoopBefore();
      [OperationContract]
      void LoopAfter();
      [OperationContract]
      void LoopIteration();
      [OperationContract]
      void StoreTmp(string tmp, string expr);
      [OperationContract]
      bool NonDeterministicIfStmtVar(string var);
      [OperationContract]
      Tuple<BigInteger, bool> NonDeterministicAssignStmtVar(string var,
                                                            string type);
      [OperationContract]
      BigInteger NonDeterministicWhileStmtBound(string var);
      [OperationContract]
      BigInteger NonDeterministicWhileStmtInit(string var);
      [OperationContract]
      void NonDeterministicWhileStmtIncrement(string var);
      [OperationContract]
      Tuple<BigInteger, bool> GetFeasibleFunctionReturnValue(string expr,
                                                             string fName,
                                                             string type);
      [OperationContract]
      void TokenLineAndColumn(string line, string column);
      [OperationContract]
      void Nop();
    }

  }
}
```

Listing 2.3: The `ICommunication` interface.

2.2 Condition solver

For solving path conditions we implemented an interface around Z3. For this, the conditions are translated from the representation using the Dafny AST to the Z3 AST nodes. They can then be handed over to the Z3 solver. The design is flexible such that is easy to change this or support different solving strategies.

2.3 Exploration strategies

Depending on how exactly the extension of the new path condition is modified, different exploration strategies are possible. We designed this as an abstract class `ExploreStrategy` that contains the two methods `nextSuccessor` and `generateSuccessorsOfCondition`. Method `nextSuccessor` returns the prefix condition that should be explored next. Method `generateSuccessorsOfCondition` generates the successors of a given condition. Using these two methods we allow the implementation of different kinds of exploration strategies. Currently Delfy contains three strategies: depth-first, breadth-first and generational. All the strategies stop going deeper in the search tree after reaching a configurable bound (Section 6.3.1 contains more details about incompleteness).

Listing 2.4 is an example that illustrates the different search strategies. It is based on the similar example in [7]. In the original example the input is a string of length 4. This string is compared to the string `bad!`. If 3 or more characters match then there occurs an error. Because Dafny does not support strings Listing 2.4 uses 4 boolean variables. If 3 or more are true an error is simulated with the statement `assert false`.

Figure 2.2 shows the search space for this program. In [7] the program is run with the string `good` as argument. In our example this corresponds to the first execution of `testme` in which all four boolean variables are set to false. This yields the path condition $\neg A \wedge \neg B \wedge \neg C \wedge \neg D$.

2.3.1 Depth-first

The `DepthFirstExploreStrategy` keeps a stack of `PathConditions` internally. When querying the next successor the path condition on top of the stack is returned. The generation of successors for a given path condition works as outlined in Algorithm 2. Line 1 specifies *pc* to be the path condition for which to generate the successors. Every path condition has a length and a prefix length. The length is the number of conditions in a particular path condition. The prefix length specifies the part of the path condition that is to be regarded as the prefix. Assuming the conditions in the path condition are numbered starting with 0, the prefix consists of the conditions with the indices 0, 1, ..., $pc.prefixlen - 1$. This corresponds to the prefix condition passed into the EXPLORE procedure in Algorithm 1. On lines $3 - 8$, Algorithm 2 then generates the successors in a depth-first manner for all the non-empty prefixes of the extension of *pc*. A new path condition that contains the prefix is created on line 4. In addition to the prefix, every extension length is considered (line 3) and the last condition of a particular extension length is negated (line 4). On line 5 the prefix length of the new path condition is then adapted to contain the whole extension. Finally, the result is pushed onto the stack that is internal to the depth-first strategy (line 6).

```
method testme(a: bool, b: bool, c: bool, d: bool)
{
  var n := 0;
  if a {
    print "b"; n := n + 1;
  } else {
    print "g";
  }
  if b {
    print "a"; n := n + 1;
  } else {
    print "o";
  }
  if c {
    print "d"; n := n + 1;
  } else {
    print "o";
  }
  if d {
    print "!"; n := n + 1;
  } else {
    print "d";
  }
  print "\n";
  if n >= 3 {
    assert false;
  }
}
```

Listing 2.4: Example to illustrate the different search strategies (based on the similar example in [7]).

Algorithm 2 Pseudo-code of successor generation for the depth-first strategy.

1: Let pc be the given path condition for which we want to generate the successors.
2: $i := pc.prefixlen$
3: **while** $i < pc.len$ **do**
4: $\quad newpc := pc[0..i-1] + +[\neg pc[i]]$
5: $\quad newpc.prefixlen := i + 1$
6: $\quad$ push $newpc$ onto the stack
7: $\quad i := i + 1$
8: **end while**

Table 2.1 shows how the stack looks like after generating the successors for the path condition $\neg A \wedge \neg B \wedge \neg C \wedge \neg D$ assuming this is the path condition after the first execution, i. e., the prefix length is 0. Note that this yields a depth-first exploration because `nextSuccessor` always returns the top of the stack. Thus, we go deeper before considering the other path conditions at the same depth.

Path condition	prefix length
$\neg A \wedge \neg B \wedge \neg C \wedge D$	4
$\neg A \wedge \neg B \wedge C$	3
$\neg A \wedge B$	2
A	1

Table 2.1: Stack after generating the successors for the search space example after the first execution in the depth-first manner. The first row represents the top of the stack.

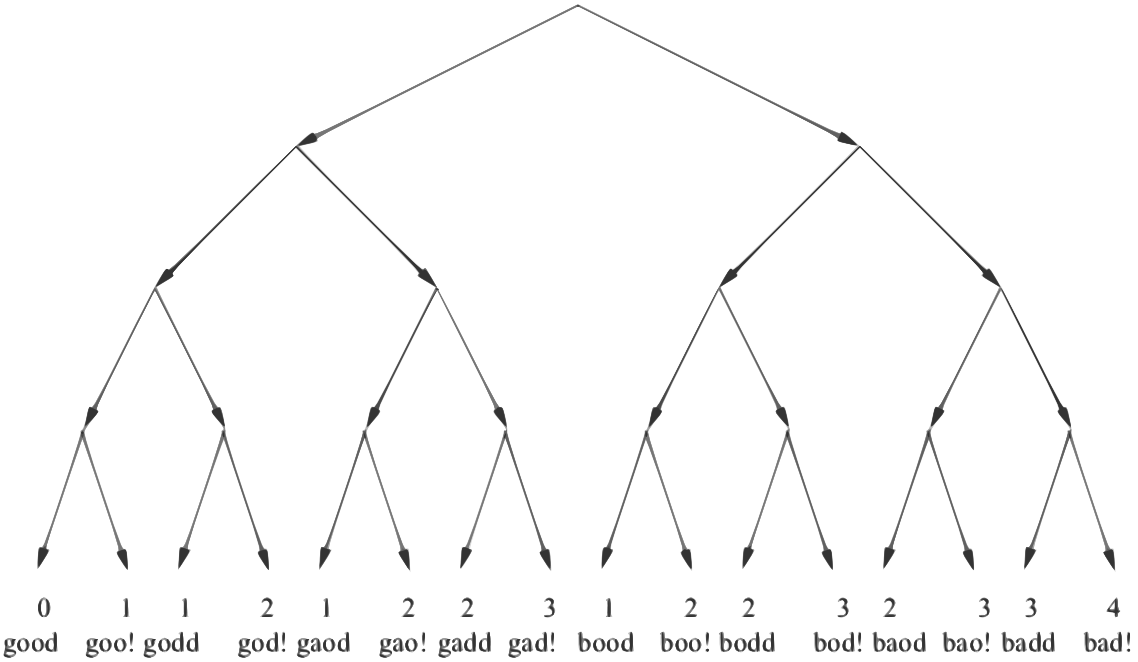

Figure 2.2: Search space visualization for the search space example in Listing 2.4; the numbers indicate the generations (reprinted from [7]).

Figure 2.3 shows the result when running Delfy for this example with the depth-first strategy. We see that the search order is indeed from left to right in the tree shown in Figure 2.2.

2.3.2 Breadth-first

To achieve a breadth-first exploration the `BreadthFirstExploreStrategy` keeps a stack of `PathConditions`. Analogous to the `DepthFirstExploreStrategy` the path condition on top of the stack is returned when querying for the next successor. But the generation of successors for a given path condition works differently. It is similar but the order in which the new conditions are generated is different. Algorithm 3 shows in pseudo-code how it is done. Line 1 specifies *pc* to be the path condition for which to generate the successors. On lines $3-8$, Algorithm 3 generates the successors in a breadth-first manner for all the non-empty prefixes of the extension of *pc*. A new path condition that contains the prefix is created on line 4. In addition to the prefix, every extension length is considered (line 3) and the last condition of a particular extension length is negated (line 4). For the breadth-first strategy i starts with $pc.len - 1$ and is decreased whereas in the depth-first strategy i starts with $pc.prefixlen$ and is increased. On line 5 the prefix length of the new path condition is then adapted to contain the whole extension. Finally, the result is pushed onto the stack that is internal to the breadth-first strategy (line 6).

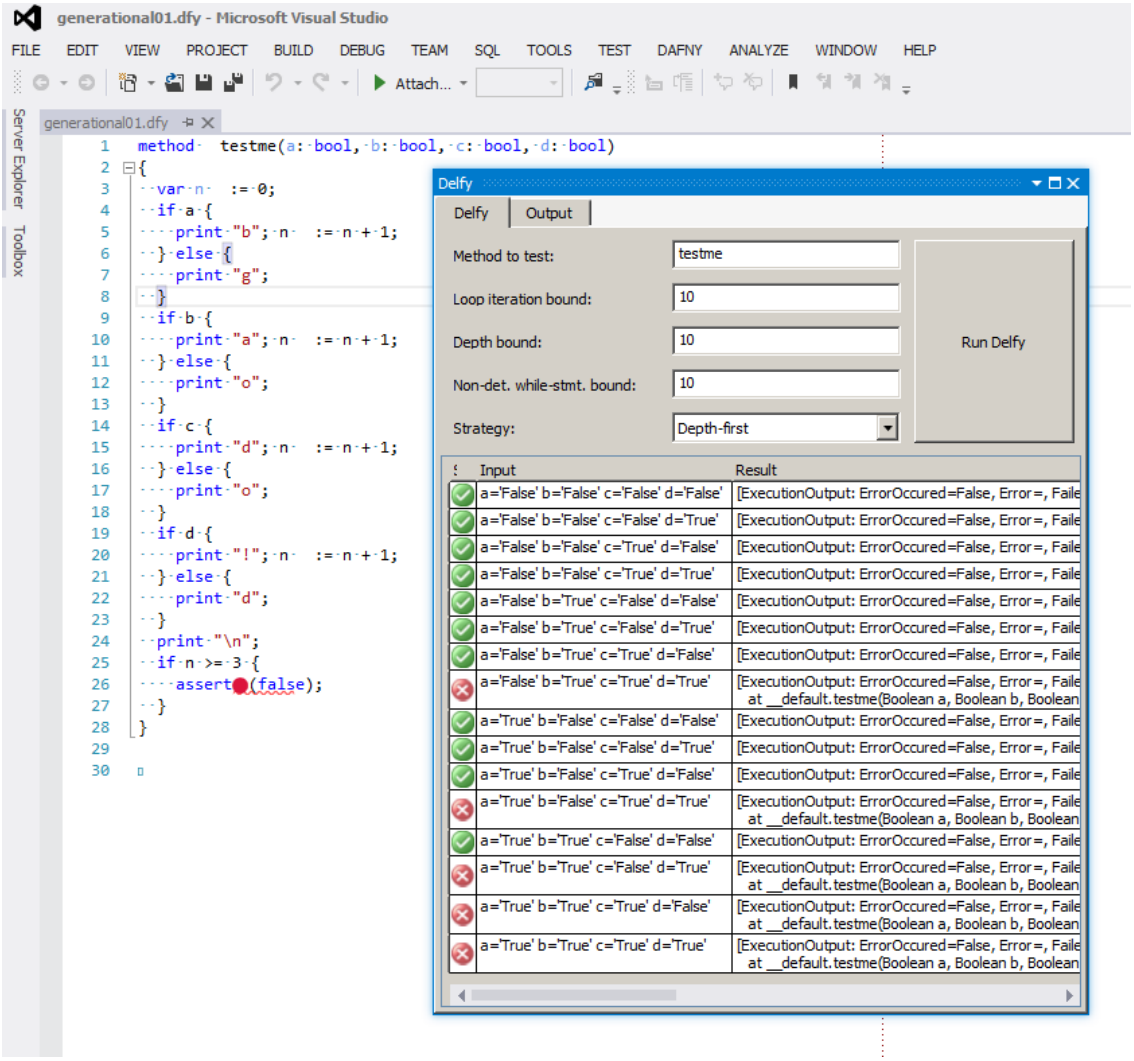

Figure 2.3: Delfy result when running the search space example with the depth-first strategy.

Algorithm 3 Pseudo-code of successor generation for the breadth-first strategy.

1: Let pc be the given path condition for which we want to generate the successors.
2: $i := pc.len - 1$
3: **while** $i >= pc.prefixlen$ **do**
4: $\quad newpc := pc[0..i-1] + +[\neg pc[i]]$
5: $\quad newpc.prefixlen := i + 1$
6: $\quad$ push $newpc$ onto the stack
7: $\quad i := i - 1$
8: **end while**

Table 2.2 shows how the stack looks like after generating the successors for the path condition $\neg A \wedge \neg B \wedge \neg C \wedge \neg D$ assuming this is the path condition after the first execution, i. e., the prefix length is 0.

Figure 2.4 shows the result when running Delfy for this example with the breadth-first strategy.

Path condition	prefix length
A	1
$\neg A \wedge B$	2
$\neg A \wedge \neg B \wedge C$	3
$\neg A \wedge \neg B \wedge \neg C \wedge D$	4

Table 2.2: Stack after generating the successors for the search space example after the first execution in the breadth-first manner. The first row represents the top of the stack.

2.3.3 Generational

As the name suggests, the idea of the generational search strategy is to search in generations. For a given path condition all the direct modifications are considered before considering indirect modifications, i. e., modifications over multiple generations [7].

The way this is implemented in `GenerationalExploreStrategy` is that we make use of a queue of `PathConditions`. The front of the queue is returned if the next successor is queried. The generation of successors works as shown in Algorithm 4. Line 1 specifies *pc* to be the path condition for which to generate the successors. On lines $3-8$, Algorithm 4 generates the successors by generations for all the non-empty prefixes of the extension of *pc*. A new path condition that contains the prefix is created on line 4. In addition to the prefix, every extension length is considered (line 3) and the last condition of a particular extension length is negated (line 4). The generational strategy works similarly to the breadth-first but the generational strategy uses a queue instead of a stack internally. On line 5 the prefix length of the new path condition is then adapted to contain the whole extension. Finally, the result is put into the queue that is internal to the generational strategy (line 6).

Algorithm 4 Pseudo-code of successor generation for the generational strategy.

```
1: Let pc be the given path condition for which we want to generate the successors.
2: i := pc.len − 1
3: while i >= pc.prefixlen do
4:     newpc := pc[0..i − 1] + +[¬pc[i]]
5:     newpc.prefixlen := i + 1
6:     enqueue newpc into the queue
7:     i := i − 1
8: end while
```

Table 2.3 shows how the queue looks like after generating the successors for the path condition $\neg A \wedge \neg B \wedge \neg C \wedge \neg D$ assuming this is the path condition after the first execution, i. e., the prefix length is 0.

Path condition	prefix length
A	1
$\neg A \wedge B$	2
$\neg A \wedge \neg B \wedge C$	3
$\neg A \wedge \neg B \wedge \neg C \wedge D$	4

Table 2.3: Queue after generating the successors for the search space example after the first execution in the generational search manner. The first row represents the front of the queue.

Figure 2.5 shows the result when running Delfy for this example with the generational strategy. We see that the search order is exactly according to the generation numbers given in the tree in

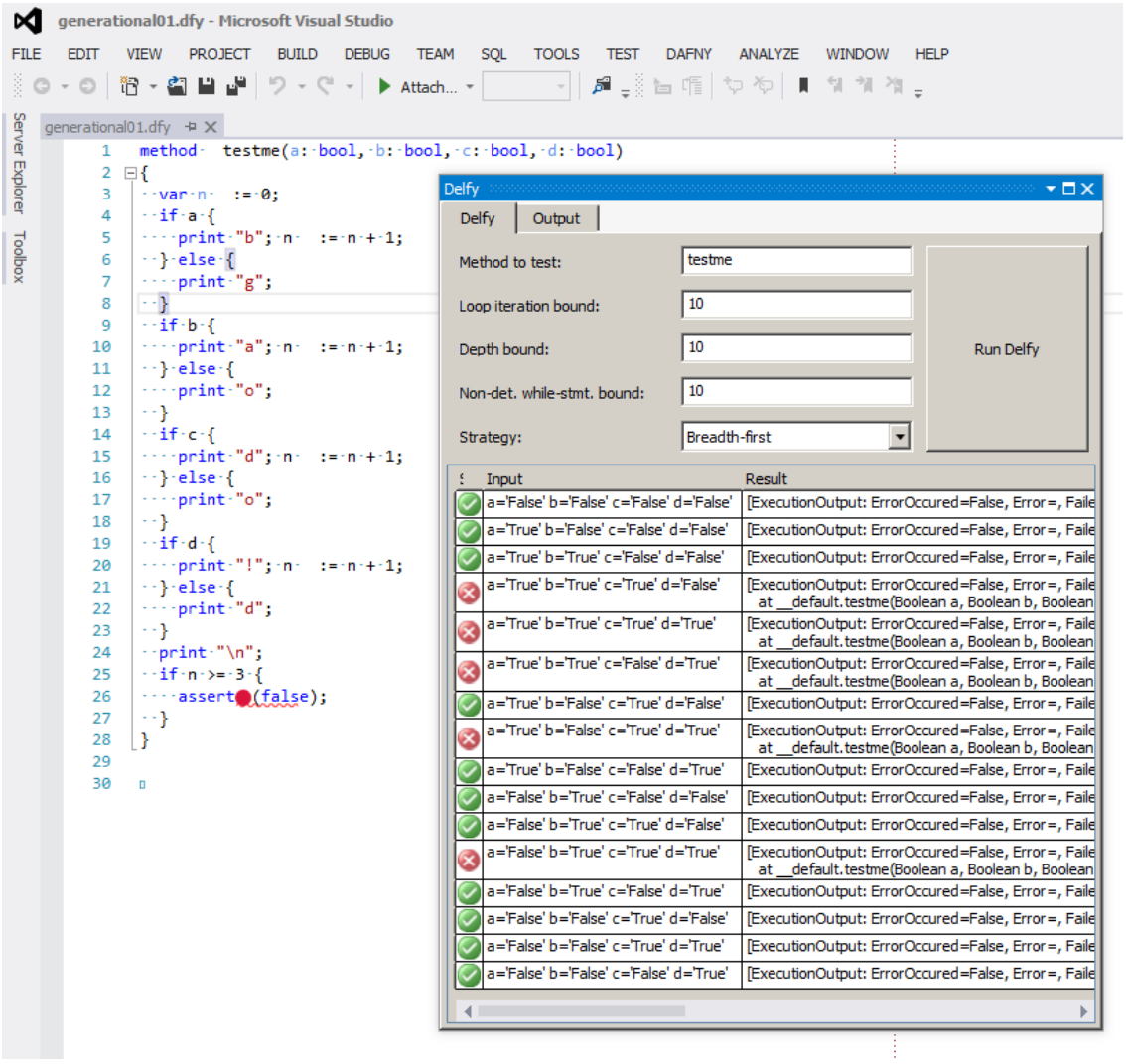

Figure 2.4: Delfy result when running the search space example with the breadth-first strategy.

Figure 2.2. This is because we use a queue and enqueue the new path conditions in the stated order.

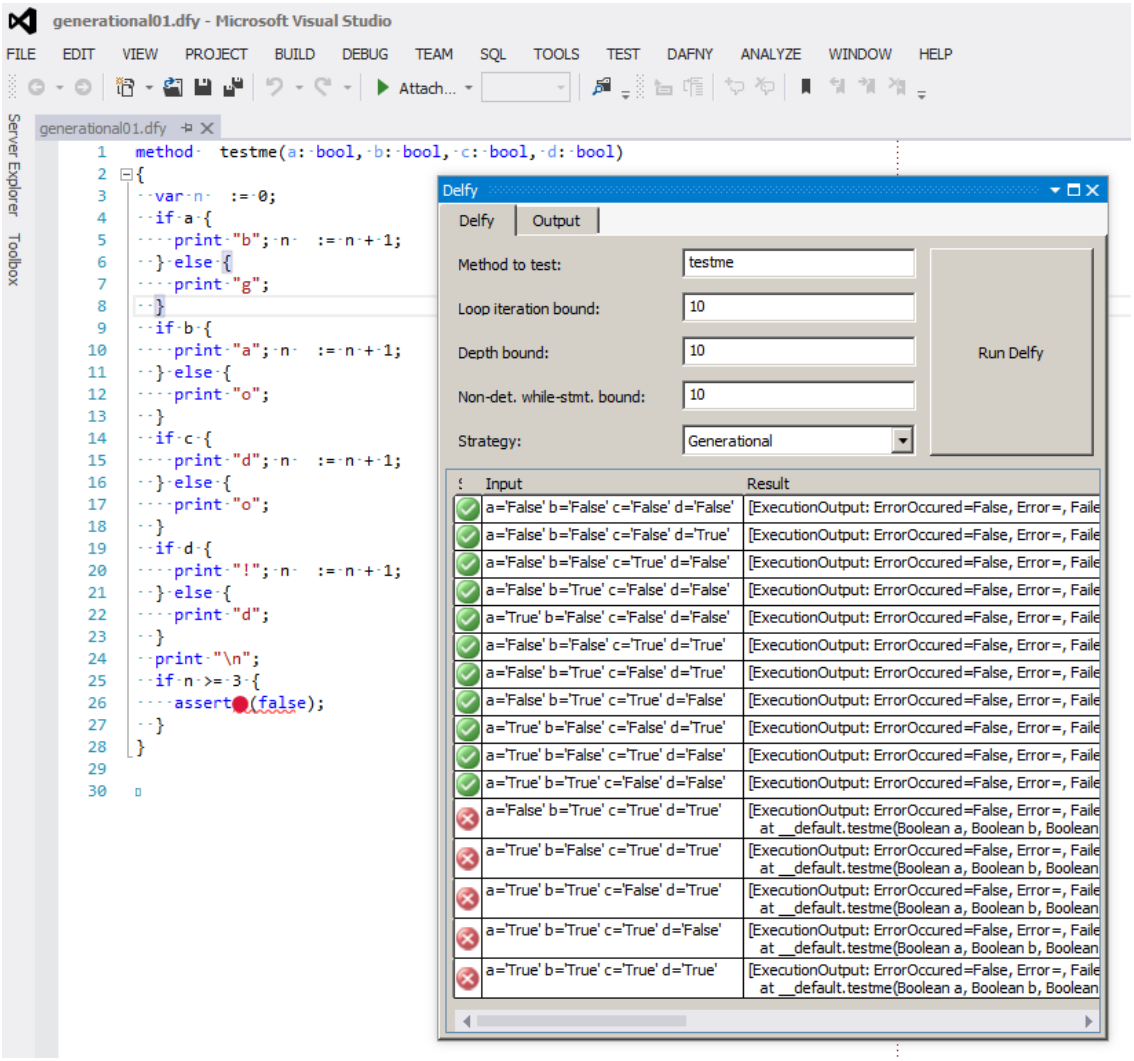

Figure 2.5: Delfy result when running the search space example with the generational strategy.

2.4 Visual Studio integration

We extended the Dafny extension [2] for Visual Studio with a graphical frontend for Delfy. Figure 2.6 shows the result of running Delfy for the example in Listing 1.1

[2]The Dafny extension source code, binaries, details and installation instructions can be found in [13].

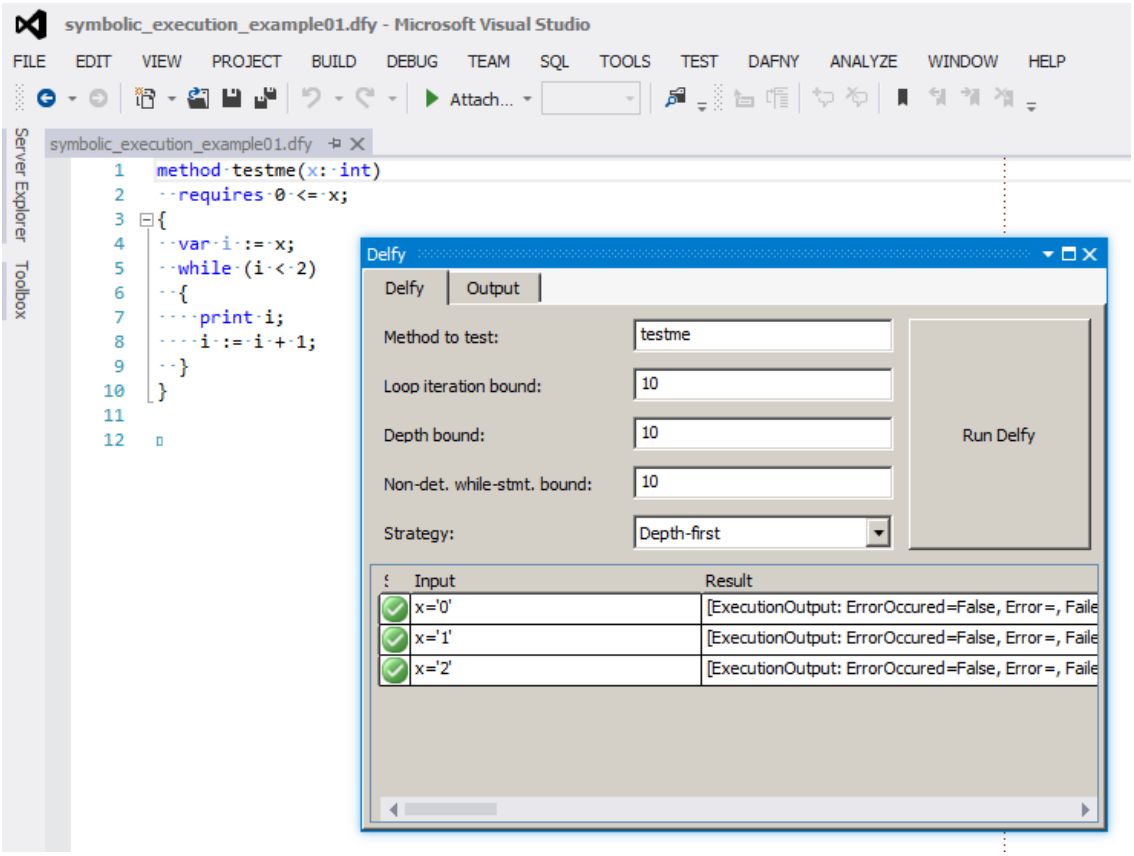

Figure 2.6: The Delfy Visual Studio extension.

Chapter 3

Support for basic Dafny features

This Chapter explains how the basic Dafny features are supported in Delfy. We describe how we extended the Dafny-to-C# compiler to support a particular feature. Moreover, we provide examples for illustration.

The symbolic state consists of a path condition *pc*, a mapping *varmap* that maps variables to symbolic expressions and a mapping *symheap* that maps references to symbolic expressions.

Furthermore, we define a substitution function *subst* that maps a symbolic expression to another symbolic expression. It substitutes variables and references in a symbolic expression by using the mapping *varmap* for variables and the mapping *symheap* for references.

3.1 Primitive Types

The Dafny primitive types

- Integers (`int`)
- Naturals (`nat`)
- Booleans (`bool`)

are supported.

Concrete execution

- `int` is translated into `BigInteger` in C#
- `nat` is translated into `BigInteger` in C#
- `bool` is translated into `boolean` in C#

Symbolic execution

- `int` is translated into a Z3 integer

- `nat` is translated into a Z3 integer with an axiom stating that a variable of type `nat` is non-negative
- `bool` is translated into a Z3 boolean

3.2 Classes and objects

Concrete execution

In the concrete execution classes and objects are mapped to C# classes and objects.

Symbolic execution

The support for classes and objects in the symbolic execution works by representing references as integers. We represent `null` with 0 and introduce axioms stating that references of different types have to be different. The Z3 back end we use can then directly handle this.

If a unit under test requires objects as inputs, the objects are then recursively constructed and initialized using the results of the symbolic execution. Section 3.2.1 discusses the challenges we faced and how we deal with them in Delfy.

3.2.1 Challenges

The creation of objects as inputs for a unit under test is a difficult task for a few reasons. Firstly, in some cases a complete object graph has to be recursively initialized. Secondly, aliasing has to be taken into account. Thirdly, objects must be initialized such that they are in a valid state.

For the construction of a complete object graph we need information in order to be able to initialize all the fields recursively. In the case of dynamic symbolic execution the solution to the path condition is the information we have available. If a field is not present at all in the condition there are a lot of possibilities to initialize it. For primitive types this could be a default value. But for reference types there are more possibilities.

References can be initialized to `null`, to a new object with some state or to an already existing object (aliasing). The first case is not difficult but the other two cases are challenging. The second case is again what is described in the previous paragraph. For the third case some pool of objects could be allocated. This pool could then be used to get references to already existing objects. The aliasing problem is for instance discussed in more detail in [8].

Even though the fields can be initialized this way there is still a problem. Just initializing the fields is not always enough. Object invariants must be taken into account. The problem and related work is discussed in [20].

In Delfy we deal with those challenges in the following way. We initialize the object graph recursively using the solution to the path condition. Fields are initialized to their default values (references to `null`) except if they occur in the solution to the path condition. If they do, we take this into account and initialize them appropriately instead of using the default value. The consequence of this is that we create new objects only if they are accessed and therefore present in the path condition. Furthermore we consider aliasing only if the path condition mentions the possibility of aliasing explicitly.

To illustrate this, Listing 3.1 shows a method `testme` that takes a `Cell` and an integer as arguments. The `Cell` class represents a cell in a linked list. It has two fields: an integer data field

and the link to the next cell. In method `testme` an error is simulated if the value of the fifth list element equals to `f(x)`. For the `next` fields there is no explicit condition. Implicit conditions are generated for the accesses when the branch condition is added to the path condition on line 14. All the possibilities are considered, i. e., every `next` field can either be `null` or non-`null`. This means that all the `next` fields are different from `null` for at least one case. Therefore, aliasing is implicitly considered. But we only consider one case. Which case this is depends on what the solver back end yields. All the `next` fields could be `null`, they could all reference the same object or some of them could reference the same object.

```
1  class Cell
2  {
3    var v: int;
4    var next: Cell;
5  }
6
7  function method f(v: int): int
8  {
9    2*v + 1
10 }
11
12 method testme(p: Cell, x: int) returns (r: int)
13 {
14   if p.next.next.next.next.v == f(x)
15   {
16     assert false; // ERROR!
17   }
18   return 0;
19 }
```

Listing 3.1: Dafny example that illustrates the problems with aliasing in object creation. The argument object `p` must be initialized recursively in order to set all the `next` fields.

3.3 Statements

Table 3.1 summarizes all the statements Delfy supports. The terminology is the one used in the Dafny AST (DafnyAst.cs). The following Sections explain how we support these in Delfy.

Statements are supported in symbolic execution through the callback "Statement".

```
void Statement(string stmt);
```

The argument is the Dafny statement that is wrapped into helper code such that the Dafny parser and resolver can be used to build the AST in Delfy.

3.3.1 `AssumeStmt`

Concrete execution

An `AssumeStmt` is translated using Code Contracts and implicit checks (Section 3.6) are generated.

Symbolic execution

An `AssumeStmt` statement

AssumeStmt
AssertStmt
PrintStmt
BreakStmt
ProduceStmt
UpdateStmt
AssignStmt
AssignSuchThatStmt
VarDecl
CallStmt
BlockStmt
IfStmt
AlternativeStmt
WhileStmt
AlternativeLoopStmt
ConcreteSyntaxStmt

Table 3.1: Dafny statements supported by Delfy.

---

assume e;

---

adds an assume condition containing $subst(e)$ to the path condition pc.

3.3.2 AssertStmt

Concrete execution

An `AssertStmt` is translated using Code Contracts and implicit checks (Section 3.6) are generated.

Symbolic execution

An `AssertStmt` statement

---

assert e;

---

adds a branch condition containing $subst(e)$ or $\neg subst(e)$ to the path condition pc depending on whether the expression holds or respectively does not hold.

3.3.3 PrintStmt

For a `PrintStmt` implicit checks (Section 3.6) are generated besides the C# code.

3.3.4 BreakStmt

For a `BreakStmt` only the C# code is generated.

3.3.5 ProduceStmt

In the following Dafny code example we have a method `alwaysOne` that returns the integer 1 unconditionally.

```
method alwaysOne() returns (result: int)
{
  return 1;
}
```

Because methods in Dafny allow multiple return values and these are named, a `ProduceStmt` is used for the return. The C# code generated for this is an assignment to the C# out parameter corresponding to `result` followed by a C# return statement.

3.3.6 UpdateStmt

Concrete execution

An update statement is translated into C# assignment statements.

Symbolic execution

An update statement

$x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n;$

updates the mappings in the symbolic state such that x_1 maps to $subst(e_1)$, x_2 maps to $subst(e_2)$, $\ldots$, x_n maps to $subst(e_n)$.

3.3.7 AssignStmt

Concrete execution

An assignment statement is translated into C# variable declarations and assignment statements.

Symbolic execution

An assignment statement

var $x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n;$

updates the mappings in the symbolic state such that x_1 maps to $subst(e_1)$, x_2 maps to $subst(e_2)$, $\ldots$, x_n maps to $subst(e_n)$.

3.3.8 AssignSuchThatStmt

The following Dafny code illustrates an `AssignSuchThatStmt`. A method `m` that takes no arguments and returns an integer is defined. It assigns an integer to a variable x such that $0 \leq x \leq 10$ and returns x.

```
method m() returns (result: int)
{
  var x :| 0 <= x <= 10;
  return x;
}
```

If the Dafny resolver is able to generate bounds, C# code is generated to find a satisfying assignment. One potential problem is that the generated code may not terminate. This can for example be the case if the right-hand-side is unsatisfiable. We handle this case by asking the condition solver whether the expression is satisfiable or not. In case it is not satisfiable we terminate the ongoing execution. Otherwise, we handle it like an `AssumeStmt` (see Section 3.3.1).

3.3.9 `VarDecl`

For a Dafny statement `VarDecl` a C# local variable declaration is generated.

3.3.10 `CallStmt`

Concrete execution

A Dafny `CallStmt` is translated into a C# call statement.

Symbolic execution

In the symbolic execution a `CallStmt`

```
var x_1, x_2, ..., x_n := m(in_1, in_2, ..., in_n);
```

or

```
x_1, x_2, ..., x_n := m(in_1, in_2, ..., in_n);
```

is supported using the two callbacks, "CallStmtBefore" and "CallStmtAfter".

```
void CallStmtBefore(string callStmt);
```

"CallStmtBefore" is called before the call is done in the concrete execution and "CallStmtAfter" after the call has returned.

```
void CallStmtAfter(string callStmt);
```

It is worth noting that a call statement can only contain a call to a method. Function calls are handled as expressions. The implication of this is that only functions can be called in expressions.

Both take the Dafny call statement wrapped inside helper code for parsing and resolving as argument. In the callback "CallStmtBefore" a new call frame is created and all the inputs in_1, $in_2, \ldots in_n$ are set. In the callback "CallStmtAfter" and all the outputs are set to the corresponding variables on the left-hand-side. The call frame is then thrown away and the previous call frame is restored.

3.3.11 `BlockStmt`

Concrete execution

For concrete execution a `BlockStmt` is translated into a C# block.

Symbolic execution

For symbolic execution the callbacks "ScopeBegin" and "ScopeEnd" are used, which do not take any arguments. "ScopeBegin" saves the current scope, creates a new scope and makes it current. "ScopeEnd" destroys the current scope and restores the saved one. This is needed to support scopes in the symbolic execution. For example, the body of a loop is a `BlockStmt`. It is possible to declare variables inside the loop that hide variables outside of the loop. The callbacks "ScopeBegin" and "ScopeEnd" are used to deal with such cases.

3.3.12 `IfStmt`

Concrete execution

For concrete execution an `IfStmt` is translated into a C# if statement.

Symbolic execution

An `IfStmt` is supported through the callbacks "Branch" and "BranchNegated".

```
void Branch(string guard);
```

```
void BranchNegated(string guard);
```

They both take as an argument the guard expression wrapped inside helper code for parsing and resolving. "Branch" creates a new branch condition with the given expression, "BranchNegated" with the negation of the given expression. This branch condition is then added to the path condition *pc*.

3.3.13 `AlternativeStmt`

An `AlternativeStmt` looks like shown in the following example.

```
if {
  case x < 0 => assert false;
  case x > 0 => print "x > 0";
  case x == 0 => print "x == 0";
  case true => print "true";
}
```

Because it is treated as a nesting of if statements it is handled analogously to the `IfStmt`.

3.3.14 `WhileStmt`

Concrete execution

For concrete execution a `WhileStmt` is translated into a C# while statement.

Symbolic execution

The `WhileStmt` is supported through the callbacks "Branch" and "BranchNegated".

"Branch" is called at the beginning of the loop body, "BranchNegated" after the loop. The semantics of "Branch" and "BranchNegated" is explained in Section 3.3.12

Moreover, the callbacks "LoopBefore", "LoopAfter" and "LoopIteration" are used in order to support breaking out of the loop after a specified amount of iterations.

There is a configurable bound on how many loop iterations are considered in the execution (Section 6.3.1 contains more details about incompleteness).

3.3.15 `AlternativeLoopStmt`

An `AlternativeStmt` looks like shown in the following example.

```
var i := 0;
while
{
  case i < 3 => i := i + 1;
  case true => break;
}
```

Due to the similarity with the `AlternativeStmt` and the `WhileStmt` it is handled similarly as they are.

There is a configurable bound on how many loop iterations are considered in the execution (Section 6.3.1 contains more details about incompleteness).

3.3.16 `ConcreteSyntaxStmt`

The `ConcreteSyntaxStmt` is an AST node that contains another statement AST node. Therefore, we need to handle the statement that is inside the `ConcreteSyntaxStmt`.

3.4 Expressions

Table 3.2 summarizes all the expressions Delfy supports. The terminology is the one used in the Dafny AST (DafnyAst.cs).

LiteralExpr
ThisExpr
IdentifierExpr
FieldSelectExpr
FunctionCallExpr
OldExpr
FreshExpr
UnaryExpr
BinaryExpr
PredicateExpr
CalcExpr
ITEExpr
ConcreteSyntaxExpression
NamedExpr

Table 3.2: Dafny expressions supported by Delfy.

In the symbolic execution the expressions are handled by querying the symbolic state.

3.4.1 FunctionCallExpr

The `FunctionCallExpr` is handled very similarly to the `CallStmt` (see Section 3.3.10). The two callbacks, "FunctionCallBefore" and "FunctionCallAfter" are used.

```
void FunctionCallBefore(string functionCallExpr);
```

```
void FunctionCallAfter(string functionCallExpr);
```

In Dafny the body of a function only contains a single expression. Thus, the semantics of handling the function call expression in the symbolic execution is to replace it with the function body in which the arguments are substituted.

3.4.2 FreshExpr

The support for `FreshExprs` is explained in Section 3.5.4.

3.4.3 OldExpr

Concrete Execution

`OldExprs` are supported using the Code Contracts support (`Contract.OldValue`). There are a few limitations. Dafny allows nested old expressions like `old(old(x))`. We do not support this in Delfy because it is not supported by Code Contracts. Moreover, for `out` parameters we use `Contract.ValueAtReturn`.

Symbolic Execution

In order to explore all outcomes of a postcondition, the callbacks "Branch" and "BranchNegated" are used. They update the path condition correspondingly. If the postcondition holds in the concrete execution, the symbolic version of the postcondition is added to the path condition, otherwise

the negated one. This means that if the concrete expression (C# level) of the postcondition is true at the end of the method body, "Branch" is used, otherwise "BranchNegated". In order to be able to know the value of the concrete postcondition, old expressions must therefore be supported outside of a `Contract.Ensures`. This is not supported by Code Contracts. Thus, we do this by remembering the expression at the start of a method/function body. We then use the remembered value whenever it is referenced inside of an old expression. For this we currently only support identifier expressions inside an old expression.

3.5 Specifications

Table 3.3 summarizes all the specification constructs Delfy supports. The terminology is the one used in the Dafny AST (DafnyAst.cs). The following Sections explain how we support these in Delfy.

`Function/Method` pre- and postconditions
`Function/Method` termination metric
`Function reads` clause
`Method modifies` clause
`FreshExpr`
Invariant for loops
Termination metric for loops

Table 3.3: Dafny specification constructs supported by Delfy.

3.5.1 `Function/Method` pre- and postconditions

Concrete execution

Pre- and postconditions are checked at the beginning and end of a function or method respectively.

Symbolic execution

For finding inputs that violate pre- and postconditions we add branch conditions to the path condition *pc* using the "Branch" and "BranchNegated" callbacks [1].

This is handled in a different way for the top-level method under test and for callees. For the top-level method under test, the precondition is assumed. It does not make sense to violate the precondition in this case because execution would stop immediately. For the postcondition we use the "Branch" and "BranchNegated" callbacks for trying to violate it. The difference for callees is that we also try to violate the precondition in addition to the postcondition.

3.5.2 Termination metrics

Concrete execution

We check that the termination metric decreases in a recursive call or from one loop iteration to the next. Moreover, we check that it is bounded.

[1] The details of the callbacks "Branch" and "BranchNegated" are explained in Section 3.3.12.

Symbolic execution

For finding inputs that violate the termination metric we add a branch condition to the path condition *pc* using the "Branch" and "BranchNegated" callbacks [2].

3.5.3 Loop invariant

Concrete execution

We check the loop invariant the same way Dafny does it, i. e., before entering the loop and at the end of the loop body.

Symbolic execution

For finding inputs that violate the loop invariant we add a branch condition to the path condition *pc* using the "Branch" and "BranchNegated" callbacks [2].

Listings 3.2 and 3.3 illustrate this. Listing 3.2 shows a Dafny method `sumN` that takes an integer n as argument and returns $s = \sum_{i=0}^{n} i$. The correct loop invariant is commented and an invariant with an error is given. This is to trigger a failure during concrete execution. Listing 3.3 shows the corresponding generated C# code. It is simplified and comments are added. The comments mark the code generated to support loop invariants in concrete and symbolic execution.

```
method sumN(n : int) returns (s : int)
  requires n >= 0;
  ensures s == (n*(n+1))/2;
{
  var i := 0;
  s := 0;
  while (i <= n)
    //invariant (s == (i*(i-1))/2) && (i <= n+1);
    invariant (s == (i*(n-1))/2) && (i <= n+1); // wrong invariant
  {
    s := s + i;
    i := i + 1;
  }
  return s;
}
```

Listing 3.2: Dafny example for a loop invariant.

3.5.4 `Modifies/Reads/Fresh` support

Checks are only needed during concrete execution because nothing is input-tainted in this case, i. e., `Modifies/Reads/Fresh` clauses cannot be influenced by different concrete input values. The checks are done by keeping sets in static fields for every function/method. The details are explained in the two Sections that follow.

[2] The details of the callbacks "Branch" and "BranchNegated" are explained in Section 3.3.12.

`Modifies/Reads` support

For every method two sets are used to support modifies clauses. One of them, say mod_{spec}, keeps track of complete object modifications and one of them, say $modField_{spec}$, keeps track of field modifications. They are initialized with all the objects/fields that are specified in the clause.

The checks that are generated make use of those sets. Upon every write access, a check is generated. This check makes sure that either the accessed object is in mod_{spec}, the accessed field is in $modField_{spec}$ or the object is fresh (the check for freshness is explained below). In addition, upon every method call a check is generated. This check makes sure that $mod_{spec}/modField_{spec}$ of the callee is a subset of $mod_{spec}/modField_{spec}$ of the caller.

Listing 3.5.4 shows a Dafny skeleton with comments to show where which checks are generated. It contains a class `C` with the two methods `F` and `G`. Both declare in a modifies clause to modify "this". `F` modifies the "data" field of `C`. The checks that are generated by the Dafny-to-C# compiler are described in the comments.

```
class C {
  // In the generated C# code, the sets Fmod_spec, FmodField_spec,
  // Gmod_spec and GmodField_spec are declared as static fields here.

  var data: int;

  method F()
    modifies this;
  {
    // In the generated C# code, Fmod_spec is initialized to contain "this",
    // FmodField_spec is initialized to
    // be empty (no field accesses in the modifies clause).
    ...
    // In the generated C# code, a check whether "this" is in Fmod_spec
    // or "this.data" is in FmodField_spec is done here.
    data = 0;
    ...
    G();
    // In the generated C# code, a check whether
    // Fmod_spec/FmodField_spec is a subset of
    // Gmod_spec/GmodField_spec, is generated here.
    ...
  }

  method G()
    modifies this;
  {
    // In the generated C# code, Gmod_spec is initialized
    // to contain "this", GmodField_spec is initialized
    // to be empty (no field accesses in the modifies clause) here.
    ...
  }
}
```

For functions this works analogously. The only difference is that functions can only have read clauses and not modifies clauses. As they are basically the same, the generated checks are very similar.

`Fresh` support

For every method, two sets, say f and f_{spec}, are used. The set f_{spec} is initialized to contain all the objects that are specified in fresh clauses inside postconditions. The set f is dynamically updated whenever a new object is created. For a fresh expression (`FreshExpr`), a check is generated. This

check is performed at the method post state and results in an error if the accessed objects in the fresh expression are not in f. For a method call, f of the caller is merged with f_{spec} of the callee. This is because according to the specification all the newly created objects in the called method are fresh in the caller as well.

Listings 3.4 and 3.5 illustrate the fresh and modifies support with an example. Listing 3.4 shows a Dafny method `CopyNode`, which makes use of a fresh clause in the postcondition. It also modifies the data field of the `node` parameter. Listing 3.5 shows the corresponding generated C# code. It is simplified and comments are added. The comments mark the code generated to support modifies and fresh clauses in concrete and symbolic execution.

```
using System.Linq;
using System.Diagnostics.Contracts;
using System.Numerics;
using System.Collections.Generic;

#region Helpers
...
#endregion

public class D {
    ...
    s = new BigInteger(0);
    var oldTmp0 = n;
    var _this = this;

    BigInteger i = new BigInteger(0);
    i = new BigInteger(0);
    DafnyToDelfy.Statement(@"i := 0");
    s = new BigInteger(0);
    DafnyToDelfy.Statement(@"s := 0");
    DafnyToDelfy.LoopBefore();

    // symbolic execution support in order to try to find inputs to break the invariant
    if ((s == (Dafny.Helpers.EuclideanDivision((i * (n - new BigInteger(1))), new BigInteger(2)))) &&
          (i <= (n + new BigInteger(1)))) {
      DafnyToDelfy.Branch(@"s == i * (n - 1) / 2 && i <= n + 1");
    } else {
      DafnyToDelfy.BranchNegated(@"s == i * (n - 1) / 2 && i <= n + 1");
    }

    // check before entering the loop
    Contract.Assert((s == (Dafny.Helpers.EuclideanDivision((i * (n - new BigInteger(1))), new
         BigInteger(2)))) && (i <= (n + new BigInteger(1))), "Loop invariant does not hold on entry."
         );

    while (i <= n)
    {
      var decr_decr2 = n - i;
      DafnyToDelfy.Branch(@"i <= n");
      DafnyToDelfy.LoopIteration();

      s = s + i;

      DafnyToDelfy.Statement(@"s := s + i;");
      i = i + new BigInteger(1);
      DafnyToDelfy.Statement(@"i := i + 1;");

      // symbolic execution support in order to try to find inputs to break the invariant
      if ((s == Dafny.Helpers.EuclideanDivision((i * (n - new BigInteger(1))), new BigInteger(2))) &&
            (i <= (n + new BigInteger(1)))) {
        DafnyToDelfy.Branch(@"(s == i * (n - 1) / 2 && i <= n + 1)");
      } else {
        DafnyToDelfy.BranchNegated(@"(s == i * (n - 1) / 2 && i <= n + 1)");
      }

      // check at the end of the loop body
      Contract.Assert((s == Dafny.Helpers.EuclideanDivision((i * (n - new BigInteger(1))), new
           BigInteger(2))) && (i <= (n + new BigInteger(1))), "Loop does not preserve the loop
           invariant.");
    }
    DafnyToDelfy.LoopAfter();
    return;
  }
}
```

Listing 3.3: The generated C# code for the example in Listing 3.2 (simplified and comments added).

```
method CopyNode(node: Node) returns (copyOfNode: Node)
  requires node != null;
  ensures copyOfNode != null;
  ensures copyOfNode.data == node.data && fresh(copyOfNode);
{
  copyOfNode := new Node.Init(node.data);
  //copyOfNode := node;
  node.data := 0;
}

method testme()
{
  var node1 := new Node.Init(0);
  var node2 := CopyNode(node1);
}

class Node
{
  var data: int;

  constructor Init(v: int)
    ensures data == v;
    modifies this`data;
  {
    data := v;
  }
}
```

Listing 3.4: Dafny example to illustrate the support for the modifies and fresh clause.

```
using System.Linq;
using System.Diagnostics.Contracts;
using System.Numerics;
using System.Collections.Generic;

#region Helpers
...
#endregion

public class D {
  public static ISet<object> _f_CopyNode = new HashSet<object>();
  public static ISet<object> _fSpec_CopyNode = new HashSet<object>();
  public static ISet<object> _mObj_CopyNode = new HashSet<object>();
  public static ISet<Frame> _mObjField_CopyNode = new HashSet<Frame>();
  public static ISet<object> _mObjSpec_CopyNode = new HashSet<object>();
  public static ISet<Frame> _mObjFieldSpec_CopyNode = new HashSet<Frame>();
  public static ISet<object> _tmp_f_CopyNode = new HashSet<object>();
  public static ISet<object> _tmp_fSpec_CopyNode = new HashSet<object>();
  public static ISet<object> _tmp_mObj_CopyNode = new HashSet<object>();
  public static ISet<Frame> _tmp_mObjField_CopyNode = new HashSet<Frame>();
  public static ISet<object> _tmp_mObjSpec_CopyNode = new HashSet<object>();
  public static ISet<Frame> _tmp_mObjFieldSpec_CopyNode = new HashSet<Frame>();
  public void @CopyNode(@Node @node, out @Node @copyOfNode)
  {
     ...
    // check fresh
    Contract.Ensures(D._f_CopyNode.IsSupersetOf(new HashSet<object>().Union(new object[] {Contract.
        ValueAtReturn(out @copyOfNode)})));
    ...
    var _this = this;

    var _nw2 = new @Node();
    D._f_CopyNode.Add(_nw2); // add _nw2 to fresh

    @_nw2.@Init((@node).@data);
    D._f_CopyNode.UnionWith(@Node._fSpec_Init);

    // check modifies regarding the "Init" call
    Contract.Assert(@Node._mObjSpec_Init.IsSubsetOf(D._mObj_CopyNode.Union(D._f_CopyNode)), "Modifies
         clause violated.");
    Contract.Assert(Contract.ForAll(@Node._mObjFieldSpec_Init, _o => D._mObj_CopyNode.Union(D.
        _f_CopyNode).Contains(_o.obj) || D._mObjField_CopyNode.Contains(_o)), "Modifies clause
        violated.");
    @copyOfNode = _nw2;
    (@node).@data = new BigInteger(0);

    // check modifies for assignment
    Contract.Assert(D._mObj_CopyNode.Union(D._f_CopyNode).Contains(@node) || D._mObjField_CopyNode.
        Contains(new Frame(@node, "data")), "Modifies clause violated.");
    D._fSpec_CopyNode.UnionWith(new HashSet<object>().Union(new object[] {@copyOfNode}));
  }
...
```

Listing 3.5: The generated C# code for the example in Listing 3.4 (simplified and comments added).

3.6 Implicit checks

For the following implicit failure possibilities we add a branch condition to the path condition *pc* using the "Branch" and "BranchNegated" callbacks [3].

- Division by zero
- Modulus by zero
- Dereference of `null`

The idea behind this is that we want to explore these failure possibilities, i. e., if it is possible to generate inputs that lead to such failures.

[3] The details of the callbacks "Branch" and "BranchNegated" are explained in Section 3.3.12.

Chapter 4

Support for other interesting Dafny features

4.1 Non-deterministic assignment statements

Listing 4.1 shows a simple Dafny example of a non-deterministic assignment statement. A variable x is declared and non-deterministically initialized. Then, it is `asserted` that x is equal to 0.

```
method testme()
{
  var x := *;
  assert x == 0;
}
```

Listing 4.1: Example of a non-deterministic assignment statement.

```
Tuple<BigInteger, bool> NonDeterministicAssignStmtVar(string var,
                                                      string type);
```

Using the callback "NonDeterministicAssignStmtVar" a new symbolic variable with the given type is dynamically introduced in the symbolic execution. It is then assigned to the left-hand side in the symbolic execution. In the concrete execution it is first initialized to what is returned by the callback. Thus, all the future uses are then taken into account in the symbolic execution. This then results in values for the non-deterministic variables according to the solution of solving the path condition. The calls to "NonDeterministicAssignStmtVar" in later concrete executions of the unit under test return then corresponding C# values for these solutions.

It is worth noting that the return type of the "NonDeterministicAssignStmtVar" callback is a tuple of a `BigInteger` and a `boolean`. We use a tuple here because the `RemoteCallback-BasedExecutor` that uses the WCF framework has problems when `object` is returned. This is sufficient though because at the time of writing this report Delfy supports the primitive types and class types for the left-hand side of a non-deterministic assignment statement variable. If the type of the left-hand side is `int` or `nat` the first item of the tuple is relevant. For type `bool`, the second item is relevant. At the time of writing this report Delfy's non-deterministic assignment statement class type support is very weak: a left-hand side of class type is always set to `null`.

4.2 Non-deterministic if statements

Listing 4.2 shows a Dafny example of a non-deterministic if statement. Method `testme` takes an integer x as argument and returns an integer. The argument x is then assigned to a newly declared variable y. Then, either 1 or 2 is assigned to y by making use of the non-deterministic if statement. Finally, y is returned.

```
method testme(x: int) returns (r: int)
{
  var y := x;
  if *
  {
    y := 1;
  }
  else
  {
    y := 2;
  }

  return y;
}
```

Listing 4.2: Example of a non-deterministic if statement.

```
bool NonDeterministicIfStmtVar(string var);
```

Using the callback "NonDeterministicIfStmtVar" a new symbolic variable is dynamically introduced in the symbolic execution. First, it is treated as being `false`. Therefore, the `else`-branch is taken and a corresponding branch condition is added to the path condition. The other branch is then explored in the symbolic execution.

4.3 Non-deterministic while statements

Listing 4.3 shows a Dafny example of a non-deterministic while statement. Method `testme` declares a variable y and initializes it to 2. It then decreases y a non-deterministic amount of times using a non-deterministic while statement. Two errors are then simulated using assertions. Both assertions might be violated because the number of loop iterations is non-deterministic. Therefore, Delfy should generate all the possible number of loop iterations to be able to cover both assertions.

```
BigInteger NonDeterministicWhileStmtBound(string var);
BigInteger NonDeterministicWhileStmtInit(string var);
void NonDeterministicWhileStmtIncrement(string var);
void NonDeterministicWhileStmtBranch(string var);
void NonDeterministicWhileStmtBranchNegated(string var);
```

The idea of supporting non-deterministic while statements is to explore all possible number of loop iterations up to a configurable bound. The approach we came up with is to set the initial loop counter value to all values smaller than the bound. Thus, execution after execution more loop iterations are explored.

Concrete execution

In the concrete execution the callbacks "NonDeterministicWhileStmtBound" and "NonDeterministicWhileStmtInit" are called to get the configurable bound and the current initial value of the

```
method testme() returns (r: int)
{
  var y: nat := 2;

  while (*)
    decreases y;
  {
    if y == 0
    {
      break;
    }

    y := y - 1;
  }

  if y == 0 { assert false; }
  if y == 1 { assert false; }

  return y;
}
```

Listing 4.3: Example of a non-deterministic while statement.

loop counter, respectively. "NonDeterministicWhileStmtIncrement" is called in every loop iteration to connect the concrete execution and the symbolic execution, i. e., to tell the symbolic execution that a loop iteration is done. The callbacks "NonDeterministicWhileStmtBranch" and "NonDeterministicWhileStmtBranchNegated" are called to update the path condition in the symbolic execution.

Symbolic execution

In the callback "NonDeterministicWhileStmtInit" a new symbolic variable representing the loop counter is dynamically created. It is then incremented in "NonDeterministicWhileStmtIncrement". The callbacks "NonDeterministicWhileStmtBranch" and "NonDeterministicWhileStmtBranchNegated" are similar to the "Branch" and "BranchNegated" callbacks as they add branch conditions to the path condition. This is needed to explore all the possible number of loop iterations in the symbolic execution.

Listing 4.4 shows the simplified C# code that is generated for the non-deterministic while statement in the example in Listing 4.3. The generated code of method `testme` first initializes the out-parameter `r` with 0 and `y` with 2. The non-deterministic while statement is supported through callbacks. Before entering the loop, a loop counter (`_nonDet1`) and a loop bound (`_tmp_2`) are initialized. The first call to "NonDeterministicWhileStmtInit" returns 0. In further calls it returns whatever is obtained through the solution of the path condition. "NonDeterministicWhileStmtBound" returns the configurable bound on the number of non-deterministic loop iterations. This means that in the first execution the loop is entered `_tmp_2` times in this example. In further executions all the possible loop iterations are considered. This is achieved with the callbacks "NonDeterministicWhileStmtBranch", "NonDeterministicWhileStmtBranchNegated" and "NonDeterministicWhileStmtIncrement". As can be seen in the example, this is done analogously to normal loops. The symbolic state is manipulated and thus dynamic symbolic execution explores all possibilities. The parameter to the callbacks specifies the name of the variable in the symbolic execution.

```
// Dafny program nondetwhilestmt01.dfy compiled into C#

using System.Linq;
using System.Diagnostics.Contracts;
using System.Numerics;
using System.Collections.Generic;
using System.Security.Cryptography;

#region Helpers
...
#endregion

public class @__default {
  ...
  public void @testme(out BigInteger @r)
  {
    @r = new BigInteger(0);
    ...
    BigInteger y = new BigInteger(0);
    y = new BigInteger(2);

    ...

    var _nonDet1 = DafnyToDelfy.NonDeterministicWhileStmtInit("_nonDet1");
    var _tmp_2 = DafnyToDelfy.NonDeterministicWhileStmtBound("_nonDet1");
    while (_nonDet1 < _tmp_2)
    {
      DafnyToDelfy.NonDeterministicWhileStmtBranch("_nonDet1");
      ...

      y = y - new BigInteger(1);

      ...

      _nonDet1 = _nonDet1 + BigInteger.One;
      DafnyToDelfy.NonDeterministicWhileStmtIncrement("_nonDet1");

      DafnyToDelfy.LoopIteration();
    }
    DafnyToDelfy.NonDeterministicWhileStmtBranchNegated("_nonDet1");
    DafnyToDelfy.LoopAfter();

    ...

    return;
  }
}
```

Listing 4.4: Instrumented C# code of the example in Listing 4.3 (simplified).

4.4 Functions with no body

```
function method FunctionWithNoBody(x: int) : int
  requires 0 < x < 3;
  ensures FunctionWithNoBody(x) == 2;

method testme() returns (r: int, s: int)
{
  s := FunctionWithNoBody(1);
}
```

Listing 4.5: Example of a function with no body.

As Dafny is a language designed for static verficiation it is possible to define functions without a body. Listing 4.5 shows an example. The function `FunctionWithNoBody` does not define a body but a precondition and a postcondition. This is useful in a verification language because there often the specifications are of interest and not the actual detailed implementation.

But this poses a problem for dynamic test generation. For example we cannot compile the C# code that results by translating the example in Listing 4.5. The reason is that if we simply generate an empty method for the function `FunctionWithNoBody` we get a compilation error because the method must return a value.

The idea we implemented is to call back into Delfy and ask for a value that satisfies the specifications of the function. Taking the example in Listing 4.5, we would generate a return statement with a call to "GetFeasibleFunctionReturnValue". At the time of writing this report Delfy only supports functions with no body with a result type of `int`, `nat` and `bool`. This is why the return type of "GetFeasibleFunctionReturnValue" is `Tuple<BigInteger, bool>`.

```
Tuple<BigInteger, bool> GetFeasibleFunctionReturnValue(string expr,
                                                       string fName,
                                                       string type);
```

In the example we would then always return 2 in a concrete execution.

4.5 Sets and sequences

4.5.1 Challenges

In order to support sets and sequences we needed to come up with a way to encode them for the prover back end. Because we only implemented one back end, namely Z3, this is what we need sets and sequences to be encoded for.

Listing 4.6 shows a Dafny program with a set membership assertion to illustrate this. In the dynamic test generation we want to generate two input sets to the `testme` method. Once the set contains the integer 1 and once it does not.

```
method testme(s: set<int>)
{
  assert 1 in s;
}
```

Listing 4.6: Example for a condition with a Dafny set membership assertion.

Our first approach was to translate the Boogie set axioms that Dafny uses for the static verification to Z3. A set is encoded as an array of booleans. An integer x is in the set if and only if the corresponding entry in the array is true. The set operations are then axiomatized.

Z3 provides interpretations, i. e., concrete assignments to variables in the condition. This is called a model. It can be queried after Z3 has been called to solve a given condition. It turned out that Z3 is not able to come up with appropriate models in all cases. In fact for the majority of cases Z3 returns the status `unknown`. Even though the status is `unknown` Z3 still provides a model but in these cases the model is simply a "candidate" model and does not have to be precise. We therefore cannot make use of this.

The axioms work for Dafny because the goal of the Dafny verifier is different than for Delfy. The Dafny verifier wants to prove an assertion. For this, the return value `unknown` is good enough because in this case Dafny can state that the assertion might not hold. But for Delfy this is different. In dynamic symbolic execution concrete input values are needed. If no concrete input values can be constructed, the unit under test cannot be concretely executed. For example, assume we have a condition that states that 1 must be in a set s. For the Dafny verifier it is enough to generate a verification condition, which can either be true or false, that encodes the condition that 1 must be in s. But Delfy actually then needs a way to construct two sets. Once with and once without a 1. For being able to do this, an appropriate model must be provided by Z3.

We ended up using the extended array theory [5] that is available in Z3. In Section 4.5.2 we explain how we make use of this.

4.5.2 Set support using Z3

Using Z3's extended array theory we are able to encode set theory with boolean algebra[1]. For this we make use of the `map` function that is provided by Z3. This function allows to apply arbitrary functions to arrays [17].

With this approach we are able to encode the set operations equality, inequality, complement, intersection, union, difference, subset and membership in the following way.

We encode the sets as arrays with a boolean element type and an index type that corresponds to the type of the set. For example, a `set<int>` would be encoded using an array mapping integers to booleans. Whether a value is in the set or not, is encoded by setting the entry of the corresponding index to true or false, respectively.

Set equality and inequality is directly supported by Z3's extended array theory, i. e., the Z3 equality and inequality operations are supported for array types.

In order to support intersection, union and complement, we make use of boolean algebra and Z3's support to map arbitrary functions to arrays. Intersection is encoded by applying boolean "and" element-wise to the elements of both arrays, union is encoded by applying boolean "or" element-wise to the elements of both arrays and complement is encoded by applying boolean "not" to all the elements of an array.

Set membership is encoded by stating that the entry for a particular index must be true in the array.

Set difference and subset are encoded using the following set identities. Let A and B be two arbitrary sets. Then, difference is encoded using the following identity: $A \backslash B = A \cap \overline{B}$. Subset is encoded using the identity $A \subseteq B \iff A = A \cap B$ and proper subset using $A \subset B \iff (A = A \cap B \wedge A \neq B)$.

[1] This approach was suggested by Leonardo de Moura (`http://stackoverflow.com/questions/18827645/is-there-a-way-to-use-z3-to-get-models-for-constraints-involving-sets`).

Set cardinality is one main operation that cannot be encoded this way. The reason for this is that there is no support in the extended array theory to determine how many entries in an array have a specific value. One possible way would be to generate a condition that tries all possibilities but the problem is that there are infinitely many. For example for one element we would need to generate a condition that tests whether the first, second, third, ... entries have the particular value. Then for all possibilities of two, three, ... elements the same must be done. This is not feasible.

Listing 4.7 shows an example of how we encode the set union operation for the case of sets of integers. Line 1 defines a parametric type "Set" of "T" to be an array with index type "T" and element type boolean. Lines $2-4$ declare three sets A, B and C as sets of integers. We want a model such that A contains the integer 1 and B contains the integer 2. This is encoded in Z3 on lines 6 and 7 by using assertions. An assertion requires a boolean argument. Thus, line 6 states that the entry of set A at index 1 must be true. Similarly, line 7 states that the entry of set B at index 2 must be true. Furthermore, we want C to be the union of A and B, i. e., $C = A \cup B$. We do this by applying the boolean function "or" to A and B. Line 8 states that C must be equal to taking the element-wise boolean "or" of A and B. Finally, line 10 tells Z3 to check satisfiability of all the conditions given using assertions. Line 11 queries the model.

Listing 4.8 shows the corresponding output of Z3. It can be seen that Z3 answers "satisfiable" (indicated by "sat" on line 1) and indeed C contains 1 and 2 in the model provided by Z3. This is shown in the model on lines $2-19$. Lines $3-8$ define A, B and C, respectively. They are just wrappers for the functions $k!0$, $k!1$ and $k!2$, respectively. On lines 10 and 11 we see that $k!0$ is true if its argument is 1 and false otherwise. This is encoded using Z3's "if-then-else" expression ("ite"). Similarly, lines 13 and 14 show that $k!1$ is true if its argument is 2 and false otherwise. And $k!2$ is true if its argument is either 2 or 1.

```
1  (define-sort Set (T) (Array T Bool))
2  (declare-const A (Set Int))
3  (declare-const B (Set Int))
4  (declare-const C (Set Int))
5
6  (assert (select A 1))
7  (assert (select B 2))
8  (assert (= C ((_ map or) A B)))
9
10 (check-sat)
11 (get-model)
```

Listing 4.7: Set union example (in the SMT 2.0 [2] specification language): $C = A \cup B$.

4.5.3 Sequence support

Sequences are encoded in Z3 as a pair containing the length of the sequence and a function that maps indices to elements. Based on this pair the sequence operations are then encoded using universal and existential quantifiers.

In the following Sections the support for the sequence operations length, equality, inequality, concatenation and membership is explained. For this, let s and t be arbitrary sequences. Then $|s|$ denotes the length of the sequence. The pair $(s_{length}, s_{elements})$ denotes the pair that encodes the sequence for Z3: s_{length} is defined as an integer constant in Z3 and represents the length of s; $s_{elements}$ denotes the function that maps the indices of s to the corresponding elements.

```
1  sat
2  (model
3    (define-fun A () (Array Int Bool)
4      (_ as-array k!0))
5    (define-fun B () (Array Int Bool)
6      (_ as-array k!1))
7    (define-fun C () (Array Int Bool)
8      (_ as-array k!2))
9    (define-fun k!0 ((x!1 Int)) Bool
10     (ite (= x!1 1) true
11       false))
12   (define-fun k!1 ((x!1 Int)) Bool
13     (ite (= x!1 2) true
14       false))
15   (define-fun k!2 ((x!1 Int)) Bool
16     (ite (= x!1 2) true
17     (ite (= x!1 1) true
18       false)))
19 )
```

Listing 4.8: The Z3 output for the example in Listing 4.7 (in the SMT 2.0 [2] specification language).

Length

In the Z3 encoding, $|s|$ is encoded as s_{length}.

Equality

The expression $s == t$ is encoded in Z3 as

$$s_{length} = t_{length} \wedge (\forall i \cdot (0 \leq i < s_{length} \wedge 0 \leq i < t_{length}) \implies (s_{elements}(i) = t_{elements}(i))).$$

Listing 4.9 shows how this looks in the SMT 2.0 [2] language. On lines 1 and 2 we declare the length constant and the elements function for the sequence s. Similarly, they are declared for the sequence t on lines 4 and 5. Lines 7 and 8 then encode the equality relation as stated in mathematical terms above. Line 9 is added to make the example more interesting. We say that the sequence s should have at least one element. Without this line a trivial solution, namely s and t are empty, is possible. Line 10 checks for satisfiability and line 11 queries the model.

Listing 4.10 shows the result of this SMT snippet. Line 1 shows that it is satisfiable (indicated by "sat"). Lines $2 - 11$ is the model. It shows that the lengths are 1 (lines $3 - 4$ and $5 - 6$) and that the elements must be the same (lines $7 - 10$). In this case, Z3 even takes the same function for both sequences.

Inequality

The expression s != t is encoded in Z3 as

$$\neg(s_{length} = t_{length} \wedge (\forall i \cdot (0 \leq i < s_{length} \wedge 0 \leq i < t_{length}) \implies (s_{elements}(i) = t_{elements}(i)))).$$

```
1  (declare-const s-length  Int)
2  (declare-fun s-elements (Int) Int)
3
4  (declare-const t-length  Int)
5  (declare-fun t-elements (Int) Int)
6
7  (assert (= s-length t-length))
8  (assert (forall ((i Int)) (=> (and (<= 0 i) (< i s-length)) (= (s-elements i) (t-
       elements i)))))
9  (assert (> s-length 0))
10 (check-sat)
11 (get-model)
```

Listing 4.9: Sequence equality example (in the SMT 2.0 [2] specification language): $s == t$.

```
1  sat
2  (model
3    (define-fun t-length () Int
4      1)
5    (define-fun s-length () Int
6      1)
7    (define-fun t-elements ((x!1 Int)) Int
8      1)
9    (define-fun s-elements ((x!1 Int)) Int
10     (t-elements x!1))
11 )
```

Listing 4.10: The Z3 output for the example in Listing 4.9 (in the SMT 2.0 [2] specification language).

Concatenation

The expression $r == s + t$, where r is a set, is encoded in Z3 as

$$r_{length} = s_{length} + t_{length} \wedge (\forall i, j \cdot (0 \leq i \wedge i < s_{length} \implies r_{elements}(i) = s_{elements}(i)) \\ \wedge (0 \leq j \wedge j < t_{length} \implies r_{elements}(s_{length} + j) = t_{elements}(j))).$$

Membership

The expression x in s, where the type of x is the type of the elements in s, is encoded in Z3 as

$$\exists i \cdot 0 \leq i < s_{length} \wedge x = s_{elements}(i).$$

Chapter 5

Static analysis for generating inputs which cover selected errors

5.1 Motivation

Delfy's dynamic test generation engine explained so far explores all the branches of a unit under test with some limitations (6.3.1). The goal is to achieve maximum branch coverage and thus to find the maximum number of errors that is possible under the limitations. However, one is not always interested in finding the highest number of errors that is possible. For example in system testing, i. e., in testing whole programs, not units, it is often not feasible to perform a full exploration of the branches. Nonetheless, it is important to know whether particular parts of a program contain errors if executed as a whole and not separately as a unit. In this case it is desirable to be able to select a specific statement and generate inputs that cover exactly this statement. But also in unit testing there are cases for which a full exploration is not feasible. Such cases benefit from the static analysis as well because not all paths must be executed.

In the context of Dafny and Delfy this is especially useful because Delfy complements Dafny's automatic verifier. An example could be that Dafny outputs that a postcondition *might* not hold. We then cannot be sure whether it is indeed violated in all cases or Dafny is just not able to tell automatically. In this case it would be useful to select this particular postcondition. Then, the idea is to use Delfy to generate inputs that cover only this postcondition. This pays off if it is not feasible for Delfy to explore the whole program.

5.2 Static symbolic execution

Our approach to generate path conditions that cover only a particular statement is static symbolic execution. The idea behind this is as it is described in Section 1.2.3. But whereas we run the unit under test concretely side-by-side with the symbolic execution in dynamic test generation (details in Section 1.2.3), in static symbolic execution we just consider the source code statically. To be more precise this means that we consider all the possible ways the symbolic state might look like after a particular statement. For every possibility we continue the symbolic execution with the next statement.

For example for the if statement

```
...
if 0 < x {
```

```
  ...
} else {
  ...
}
...
```

the two possibilities $0 < x$ and $\neg(0 < x)$ are considered because statically we do not know which one holds in a concrete execution. In the first case we add $0 < x$ to the path condition and update the symbolic state according to the statements in the then-branch. In the second case we add $\neg(0 < x)$ to the path condition and update the symbolic state according to the statements in the else-branch. All the possible resulting symbolic states are then considered in the statements that follow this if statement.

5.2.1 Control Flow Graph (CFG)

A control flow graph is built in order to know which paths need to be considered to reach the particular statements. First, the statements to cover are found in a top-down manner. While walking down the AST looking for the target statements, the CFG is built. By building the CFG in this way, we only build the necessary parts of the CFG. I. e., nodes that are further down in the AST than the target nodes are not considered at all. Then, all the target nodes are considered one after the other as starting points for a bottom-up traversal of the CFG. These bottom-up traversals mark the Dafny AST nodes for the static symbolic execution. The nodes are marked in such a way that the static symbolic execution knows which branches need to be taken into account to reach the target statements. As an example for an if statement we mark whether we need to generate a path condition for the then-branch, for the else-branch or for both. This information is then used during the static symbolic execution to decrease the number of path conditions we generate.

5.2.2 Design and Implementation

For the static symbolic execution we designed a new class `StaticSymbolicExecutor`. This class is able to generate all the possible symbolic states given a Dafny program and a target location. It performs a static symbolic execution on the Dafny program AST. In particular it therefore generates all possible path conditions that cover the given location. Every such path condition can then be handed over to the Delfy dynamic test generation engine. Delfy is then able to solve the conditions as usual and concretely run the program with the corresponding concrete inputs.

The Dafny features are handled the same as they are in the symbolic execution of the dynamic test generation engine. Details can therefore be found in Chapters 3 and 4 in the Subsection "Symbolic execution" of a particular feature.

5.2.3 Challenges

State explosion

In static symbolic execution we perform model checking. Model checking is a technique for automatically verifying hardware and software systems. It represents a system as a finite set of states and explores the state space. The size of the state space grows exponentially with the number of state variables. This is called the "state explosion problem" [3].

In our case this means that there are cases for which we generate exponentially many possible path conditions. We deal with this problem in two ways. Firstly, we take the information the CFG provides and generate only those path conditions that actually lead to errors. Secondly, we try to be conservative. This means that during static analysis we prune the AST in order to avoid generating too many path conditions. This means that after we built the CFG, when we walk down the AST with the marked nodes, we not only stop at target statements but also statements which we consider to be too expensive to fully cover in the static analysis, e. g. while statements. The reason for this is that already a small number of loop iterations - especially of nested loops - yields too many path conditions. But when we then hand over the generated path conditions to the dynamic test generation engine, the generated path conditions are treated as prefixes. This means that all the parts that were pruned during the static analysis are explored during dynamic test generation.

Listing 5.1 shows an example of a Dafny program for which there are exponentially many possible path conditions for reaching an error statement, in particular to reach the last assertion in the method body. For every loop iteration of the outer loop in the "`testme` method" there are three possible loop iterations of the inner loop. Moreover, for every such path condition there are two more for the first, second, third, ... if statements. All these numbers multiply. Thus, this example illustrates that already a small bound on the number of non-deterministic loop iterations and a small number of if statements can mean that the static analysis is not feasible without pruning.

```
method testme(aa: int, bb: int) returns (a: int, b: int)
{
  a, b := aa, bb;
  while *
    decreases 2 - a;
  {
    if a >= 2 { break; }

    a := a + 1;

    b := 0;
    while *
      decreases 2 - b;
    {
      if b >= 2 { break; }
      b := b + 1;
    }
  }

  if a == 0 && b == 0 { assert false; }
  if a == 1 && b == 0 { assert false; }
  if a == 1 && b == 1 { assert false; }
  if a == 1 && b == 2 { assert false; }
  if a == 2 && b == 0 { assert false; }
  if a == 2 && b == 1 { assert false; }
  if a == 2 && b == 2 { assert false; }

  assert false;
}
```

Listing 5.1: Dafny example with exponentially many path conditions for reaching the last statement of the method body.

5.3 Visual Studio integration enhancement

We extended Visual Studio extension that is explained in Section 2.4. If Dafny shows an error this error can be selected. If Delfy is run and an error is selected, the static analysis generates path conditions only for the selected error location. These are then handed over to the dynamic test generation engine. As a further step, they are solved and the unit under test is concretely run for the particular concrete inputs.

Figure 5.1 shows the result of running Delfy after an error has been selected. The example contains a method `testme`, which takes two integers, x and y, as arguments. If x is greater than zero, an error is simulated by the statement `assert false`. Similarly, if y is greater than zero, an error is simulated. The Dafny extension shows that both assertions might not hold. This is indicated by the squiggly red lines and the red dots. We selected the assertion that is reached if y is greater than zero and ran Delfy. The inputs that Delfy generates show that only one input is generated. It covers the selected error with the concrete values $x = 0$ and $y = 1$.

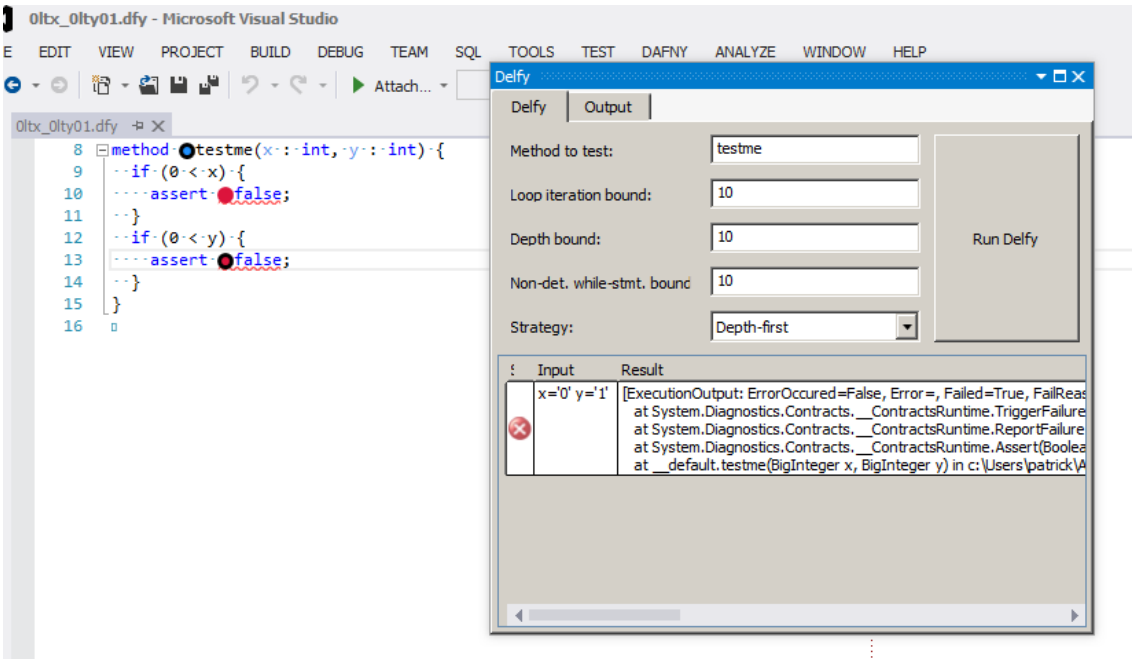

Figure 5.1: The Delfy Visual Studio extension with a selected error.

Chapter 6

Results

6.1 Test suite

In order to evaluate the Delfy tool that was built during this Master's thesis project we created a test suite containing Dafny programs. The main Dafny code base that we have available is the test suite of the Dafny project itself. In addition to those test cases the test suite we used for evaluating Delfy contains all the tests that we created during the design and implementation phase of the project. The test suite contains 391 Dafny programs.

6.2 Evaluation

6.2.1 Feature support

We investigated the test cases in order to see how many Delfy is able to handle. 68 of the 391 Dafny programs in the test suite cannot be handled by Delfy because of missing feature support. Thus, about 82% of the test cases in the test suite can be explored using Delfy. Section 6.3.2 contains an overview of the unsupported features.

6.2.2 Comparison of dynamic test generation with and without static analysis (SA)

We implemented the static analysis in order to be able to only generate inputs that cover particular statements. The main idea is to only cover statements for which the Dafny verifier reports an error.

In this Section we describe the experiments we did to see whether this is helpful in reducing the number of inputs we generate. We then present our results.

We conducted the experiments in the following way. We did two different kinds of experiments. One, in which we selected tests from the test suite that we then adapted before running Delfy. And a second one, in which we ran Delfy for all the tests in the test suite as they are.

For the first experiment we took a method to test from the test suite. We then created three different versions of this test. The first one verifies. For the other two tests we introduced different errors. The errors were introduced in verification constructs by removing them completely or partially. For example, a modifies clause can be completely removed to create a meaningful error.

We then ran Delfy with and without static analysis on the selected and adapted tests in order to compare the number of generated inputs. Table 6.1 summarizes the results of the experiments. The first column contains the name of the test case. The second column the description of the different versions of this test case and the other two columns the number of inputs generated with and without static analysis, respectively. This clearly shows that if there are no errors in a test it is always better with the static analysis. The reason is that we then generate no path conditions at all. Therefore, we do not run Delfy and so we do not generate any inputs. For the other cases the static analysis is not always of much help. The reason is that the static analysis is conservative as explained in Section 5.2.3. In order for the static analysis to be fast we prune the AST. So for example if a test contains a loop as the first statement then the static analysis does not help because we are conservative and prune the loop during the static analysis.

Test case	Changes	# w/ SA	# w/o SA
`AdvancedLHS.dfy`	verified	0	2
	introduced error in modifies clause	0	0
	introduced an additional assertion error	0	2
`BinarySearch01.dfy`	verified	0	17
	introduced error in postcondition	16	17
	introduced error in loop invariant	17	17
`KatzManna01.dfy`	verified	0	1
	introduced error in loop invariant	22	22
	introduced error in loop termination	12	12
`KatzManna02.dfy`	verified	0	11
	introduced error in loop invariant	8	9
	introduced error in loop termination	4	5
`KatzManna03.dfy`	verified	0	0
	introduced error in modifies clause	0	0
	introduced error in loop invariant	0	0
`loopinvariants03.dfy`	verified	0	4
	introduced error in loop invariant	3	3
	introduced error in postcondition	3	3
`modifies01.dfy`	verified	0	1
	introduced error in fresh expression	1	1
	introduced error in modifies clause	1	1
`reads01.dfy`	verified	0	1
	introduced error in reads clause	0	1
	introduced another error in reads clause	0	1
`seq02.dfy`	verified	0	4
	introduced assertion error	1	4
	introduced two additional assertion errors	4	4
`set08.dfy`	verified	0	1
	introduced assertion error	1	1
	introduced an additional assertion error	2	3

Table 6.1: Comparison results of the dynamic test generation with and without static analysis. Every line corresponds to one experiment. The first column states the name of the test case. The second column explains the test, i. e., whether it verifies or the changes we made. The other two columns contain the number of generated inputs with and without static analysis, respectively. The loop iteration bound was set to 10 and the non-deterministic loop iteration bound to 2. The generational search strategy was used with 3 as the bound on the number of generations (search depth).

For the second experiment we ran Delfy with and without static analysis on the whole test suite, i. e., on all the tests as they are in the test suite. The results of this are summarized in Table 6.2. The first column contains the name of the test case. The second and third column show the number of inputs generated with and without static analysis, respectively. The rows in which both these columns contain a zero are the test cases that cannot be handled by Delfy due to unsupported features. What this table shows us is that using the static analysis helps in most cases. This is because it avoids generating successful inputs, i. e., inputs that do not cover an unverified statement.

Test case	# w/ SA	# w/o SA
0ltx_0lty01.dfy	2	3
0ltx_0lty02.dfy	2	3
alternativeloopstmt01.dfy	0	4
alternativestmt01.dfy	1	3
assert01.dfy	1	2
assert02.dfy	1	2
assert03.dfy	1	2
assert04.dfy	1	2
assert05.dfy	1	2
assignment01.dfy	1	2
assignment02.dfy	1	2
assignment03.dfy	0	1
assignsuchthat01.dfy	0	1
assignsuchthat02.dfy	0	1
assume01.dfy	0	1
assume02.dfy	0	4
class01.dfy	1	2
class02.dfy	1	3
class03.dfy	1	2
class04.dfy	1	2
class05.dfy	1	3
class06.dfy	2	4
class07.dfy	1	2
classargument01.dfy	1	2
classargument02.dfy	2	3
classargument03.dfy	1	4
classargument04.dfy	2	4
classargument05.dfy	1	3
classargument06.dfy	1	5
classargument07.dfy	4	6
classargument08.dfy	2	3
classargumentscope01.dfy	0	1
classbinarysearchtree01.dfy	5	6
division01.dfy	1	2
exists01.dfy	1	2
fibonacci01.dfy	0	4
fibonacci02.dfy	0	4
forall01.dfy	1	2
function01.dfy	2	5
function02.dfy	2	5
function03.dfy	2	4
functiontermination01.dfy	0	4

functiontermination02.dfy	0	4
functiontermination03.dfy	0	4
functiontermination04.dfy	0	2
functionwithnobody01.dfy	0	1
gcd01.dfy	1	11
gcd02.dfy	2	3
gcd03.dfy	2	3
gcd04.dfy	1	3
generational01.dfy	1	15
ifthenelseexpression01.dfy	1	2
ifthenelseexpression02.dfy	1	2
ifthenelseexpression03.dfy	1	3
integers01.dfy	1	2
integers02.dfy	1	2
intermediatepresentation01.dfy	0	4
intermediatepresentation02.dfy	0	4
intermediatepresentationfreshexpr01.dfy	0	1
intermediatepresentationfunctiontermination01.dfy	0	4
intermediatepresentationfunctiontermination02.dfy	0	2
intermediatepresentationmodifies01.dfy	1	1
intermediatepresentationnondetwhilestmt01.dfy	2	3
intermediatepresentationreads01.dfy	0	1
intermediatepresentationwhilestmtinvariant01.dfy	1	3
intermediatepresentationwhilestmttermination01.dfy	1	1
linear01.dfy	1	4
listinsert01.dfy	1	1
loopinvariants01.dfy	0	7
loopinvariants02.dfy	1	8
loopinvariants03.dfy	0	4
loopinvariants04.dfy	1	3
looptermination01.dfy	1	4
looptermination02.dfy	1	6
looptermination03.dfy	1	1
map01.dfy	0	1
methodtermination01.dfy	0	4
methodtermination02.dfy	0	4
methodtermination03.dfy	0	4
methodtermination04.dfy	1	2
modifies01.dfy	0	1
modifies02.dfy	0	1
modifies03.dfy	1	1
modulus.dfy	1	2
multiassignment01.dfy	1	1
multiassignment02.dfy	0	4
multiset01.dfy	0	1
naturals01.dfy	0	1
naturals02.dfy	1	2
nondeterministicassignstmt01.dfy	1	2
nondeterministicassignstmt02.dfy	1	3
nondeterministicassignstmt03.dfy	1	2
nondeterministicassignstmt04.dfy	1	4

nondeterministicifstmt01.dfy	0	2
nondeterministicifstmt02.dfy	0	4
nondeterministicifstmt03.dfy	2	2
nondeterministicifstmt04.dfy	2	2
nondeterministicifstmt05.dfy	3	3
nondeterministicifstmt06.dfy	1	3
nondeterministicifstmt07.dfy	4	5
nondeterministicifstmt08.dfy	1	4
nondeterministicwhilestmt01.dfy	2	3
nondeterministicwhilestmt02.dfy	1	5
nondeterministicwhilestmt03.dfy	1	2
nondeterministicwhilestmt04.dfy	1	1
nondeterministicwhilestmt05.dfy	1	3
nondeterministicwhilestmt06.dfy	1	13
nondeterministicwhilestmt07.dfy	1	7
nonlinear01.dfy	1	3
null01.dfy	1	2
null02.dfy	1	2
null03.dfy	2	3
null04.dfy	1	3
prepost01.dfy	1	3
prepost02.dfy	0	2
prepost03.dfy	0	1
prepost04.dfy	0	1
prepost05.dfy	0	4
prepost06.dfy	1	5
reads01.dfy	0	1
reads02.dfy	0	1
reads03.dfy	0	1
reads04.dfy	0	1
reads05.dfy	0	1
scope01.dfy	0	1
seq01.dfy	0	1
seq02.dfy	1	4
seq03.dfy	1	2
seq04.dfy	1	2
seq05.dfy	1	4
seq06.dfy	0	1
seq07.dfy	0	1
seq08.dfy	1	2
seq09.dfy	1	2
seq10.dfy	1	1
seq11.dfy	1	3
seq12.dfy	0	1
seq13.dfy	1	2
seq14.dfy	0	2
seq15.dfy	1	2
seq16.dfy	0	0
seq17.dfy	0	0
seqconcatenation01.dfy	1	2
seqconcatenation02.dfy	1	2

seqconcatenation03.dfy	1	2
seqconcatenation04.dfy	1	2
seqconcatenation05.dfy	1	1
seqconcatenation06.dfy	1	1
seqcontains01.dfy	1	2
seqcontains02.dfy	1	2
seqcontains03.dfy	1	2
seqcontains04.dfy	1	2
seqcontains05.dfy	1	2
seqcontains06.dfy	1	2
seqcontains07.dfy	1	2
seqcontains08.dfy	1	2
seqequality01.dfy	1	2
seqequality02.dfy	1	2
seqequality03.dfy	1	2
seqequality04.dfy	1	2
seqlength01.dfy	1	2
seqlength02.dfy	1	2
seqselect01.dfy	1	2
seqselect02.dfy	1	2
set01.dfy	0	1
set02.dfy	1	2
set03.dfy	1	2
set04.dfy	1	2
set05.dfy	1	4
set06.dfy	1	4
set07.dfy	0	1
set08.dfy	1	1
set09.dfy	1	2
set10.dfy	0	0
set11.dfy	0	0
setdifference01.dfy	1	2
setdifference02.dfy	1	2
setequality01.dfy	1	2
setequality02.dfy	1	2
setequality03.dfy	1	2
setequality04.dfy	1	2
setintersection01.dfy	1	2
setintersection02.dfy	1	2
setmembership01.dfy	1	2
setmembership02.dfy	1	2
setmembership03.dfy	1	2
setmembership04.dfy	1	2
setmembership05.dfy	1	2
setmembership06.dfy	1	2
setmembership07.dfy	1	2
setmembership08.dfy	1	2
setpropersubset01.dfy	1	2
setpropersubset02.dfy	1	2
setsubset01.dfy	1	2
setsubset02.dfy	1	2

setunion01.dfy	1	2
setunion02.dfy	1	2
squareRoot.dfy	1	2
test001.dfy	1	2
test002.dfy	1	2
test003.dfy	1	3
test004.dfy	1	2
test005.dfy	2	4
test006.dfy	1	3
test007.dfy	1	15
test008.dfy	1	2
test009.dfy	1	1
test010.dfy	2	6
test011.dfy	1	3
test012.dfy	1	2
unary01.dfy	1	2
unary02.dfy	0	1
zunebug01.dfy	1	4
Add01.dfy	1	2
Add02.dfy	1	2
AdvancedLHS.dfy	0	2
AlternativeStatements.dfy	1	6
BDD.dfy	0	0
BinarySearch01.dfy	1	17
BinarySearch02.dfy	1	17
BinaryTree.dfy	0	0
BreadthFirstSearch.dfy	0	0
Breaks01.dfy	0	7
Breaks02.dfy	2	6
CachedContainer.dfy	0	0
CalcExample.dfy	0	0
Calculations.dfy	0	0
Calls.dfy	1	2
Celebrity.dfy	0	0
ChainingOperators.dfy	1	1
Classics.dfy	0	1
Classics01.dfy	0	1
Classics02.dfy	0	4
Combinators.dfy	0	0
Composite.dfy	0	0
COST-verif-comp-2011-1-MaxArray.dfy	0	0
COST-verif-comp-2011-2-MaxTree-class.dfy	0	0
COST-verif-comp-2011-2-MaxTree-datatype.dfy	0	0
COST-verif-comp-2011-3-TwoDuplicates.dfy	0	0
COST-verif-comp-2011-4-FloydCycleDetect.dfy	0	0
Cube.dfy	1	1
Cubes.dfy	0	1
Dijkstra.dfy	0	0
ExtensibleArray.dfy	0	0
ExtensibleArrayAuto.dfy	0	0
Filter.dfy	0	0

FindZero.dfy	0	1
FunctionSpecifications01.dfy	1	4
FunctionSpecifications02.dfy	1	4
FunctionSpecifications03.dfy	1	4
FunctionSpecifications04.dfy	1	3
FunctionSpecifications05.dfy	1	3
FunctionSpecifications06.dfy	0	0
FunctionSpecifications07.dfy	0	0
FunctionSpecifications08.dfy	0	0
Induction01.dfy	1	4
Induction02.dfy	0	4
Induction03.dfy	0	4
Induction04.dfy	0	4
InductionVsCoinduction.dfy	0	0
InfiniteTrees.dfy	0	0
Intervals.dfy	0	0
KatzManna01.dfy	0	1
KatzManna02.dfy	0	11
KatzManna03.dfy	1	1
LazyInitArray.dfy	0	1
ListContents.dfy	0	0
ListCopy.dfy	0	0
ListReverse.dfy	0	0
MajorityVote.dfy	0	0
MatrixFun.dfy	0	1
MonotonicHeapstate.dfy	0	1
MoreInduction.dfy	0	0
MultiAssignments01.dfy	2	3
MultiAssignments02.dfy	2	6
Paulson.dfy	0	0
pow2.dfy	0	4
PriorityQueue.dfy	0	1
Problem1-SumMax.dfy	0	1
Problem2-Invert.dfy	0	0
Problem3-FindZero.dfy	0	0
Problem4-Queens.dfy	0	0
Problem5-DoubleEndedQueue.dfy	0	0
Queue01.dfy	0	0
Queue02.dfy	0	0
ReturnTests01.dfy	0	1
ReturnTests02.dfy	0	1
ReturnTests03.dfy	0	1
ReturnTests04.dfy	0	1
ReturnTests05.dfy	0	1
ReturnTests06.dfy	0	1
RingBuffer.dfy	0	1
RingBufferAuto.dfy	0	0
Rippling.dfy	0	0
SchorrWaite-stages.dfy	0	0
SchorrWaite.dfy	0	0
SegmentSum01.dfy	1	2

SegmentSum02.dfy	0	10
SeparationLogicList.dfy	0	0
Simple.dfy	0	0
SimpleCoinduction.dfy	0	0
SimpleInduction.dfy	0	0
Skeletons.dfy	0	0
SmallTests.dfy	0	0
SnapshotableTrees.dfy	0	0
SparseArray.dfy	0	1
SplitExpr.dfy	0	1
StatementExpressions.dfy	0	1
StoreAndRetrieve.dfy	0	0
Streams.dfy	0	0
Substitution.dfy	0	0
SumOfCubes.dfy	0	1
Superposition.dfy	0	1
TailCalls.dfy	0	0
Termination.dfy	0	0
TerminationDemos01.dfy	0	4
TerminationDemos02.dfy	1	3
TerminationDemos03.dfy	2	5
TerminationDemos04.dfy	0	0
Tree.dfy	0	0
TreeBarrier.dfy	0	0
TreeDatatype.dfy	0	0
TreeFill.dfy	0	0
TuringFactorial.dfy	0	4
Two-Way-Sort.dfy	0	1
TypeAntecedents.dfy	0	0
TypeParameters.dfy	0	0
TypeTests.dfy	0	0
UltraFilter.dfy	0	0
UnboundedStack.dfy	0	1
vsi_b1.dfy	0	0
vsi_b2.dfy	0	1
vsi_b3.dfy	0	0
vsi_b4.dfy	0	0
vsi_b5.dfy	0	0
vsi_b6.dfy	0	0
vsi_b7.dfy	0	0
vsi_b8.dfy	0	0
WideTrees.dfy	0	1
Zip.dfy	0	1

Table 6.2: Comparison results of the dynamic test generation with and without static analysis for the tests in the test suite. The first column states the name of the test case. The other two columns contain the number of generated inputs with and without static analysis, respectively. The loop iteration bound was set to 10 and the non-deterministic loop iteration bound to 2. The generational search strategy was used with 3 as the bound on the number of generations (search depth).

6.2.3 Comparison with the Boogie Verification Debugger (BVD)

The Boogie Verification Debugger is a tool to help users understand the potential program errors reported by a program verifier. Although the user interface is similar to a dynamic debugger, the debugging happens statically. The different levels of abstraction between the theorem prover counterexample and the program make such a debugger difficult to construct. BVD instruments the verification conditions it hands over to the theorem prover. It is then able to reconstruct states and memory values that the user can understand [10].

In the process of manually inspecting the results of Delfy for different Dafny programs we also used BVD. We recognized patterns in the tests. There are patterns for which a concrete input leading to a failure is enough to understand the problem. But there are also patterns for which in addition to a concrete input BVD - in comparison to Delfy - helps to gain a better understanding of what is going on. Moreover, there are cases for which BVD does not help.

Object graphs

We observed that if a unit under test involves a lot of heap structure BVD is more helpful than Delfy. This is because it allows to view the state of these object graphs whereas Delfy only provides values that are constrained through the path condition.

Figure 6.1 shows an example of this. There is a method `testme` that we want to test. It takes as parameters a `Cell` and an integer. The `Cell` class can be thought of a linked structure such as a node in a linked list. It has two fields, a data field `v` and a next field `next`. Inside `testme` we create a new `Cell` and then there are some conditions under which the `assert false` is reached. Both, Delfy as well as BVD provide inputs that violate the assertion. BVD provides more information though. It shows the state of the objects.

Thus, this example illustrates well the difference between Delfy and BVD from a "user" perspective. Delfy gives a concrete input that leads to a failure. This is helpful for reproducing errors. But to actually debug the error and to understand the reason for the error, the information BVD provides is crucial in this case of object graphs.

Method calls

In the experiments in which we tested a method that calls other methods or functions we noted that BVD does not relate the parameters back to the ones of the unit under test. Therefore, in this case Delfy proves more useful as the illustrative example in Figure 6.2 shows. The method `testme` takes an integer `x` as argument. It then calls the method `Theorem0` with this argument. Inside `Theorem0` there is an assertion that does not always hold as indicated by the Dafny verifier.

Because BVD is a static debugger there is no runtime call stack. Calls are encoded into Boogie. The values are not traced back.

Loops

The information BVD provides for cases with loops is not helpful as the example in Figure 6.3 shows. It is a simple method for calculating the greatest common divisor (GCD) of two integers. In the end we simulate an error if the GCD is 17. One can see that Delfy provides inputs that exercise this path. In contrast BVD is not able to deal with the loop. It just states that the GCD must be 17. But with inputs 16 and 1 this is not possible.

BVD is a static debugger. But in order to step through a loop the state of every loop iteration

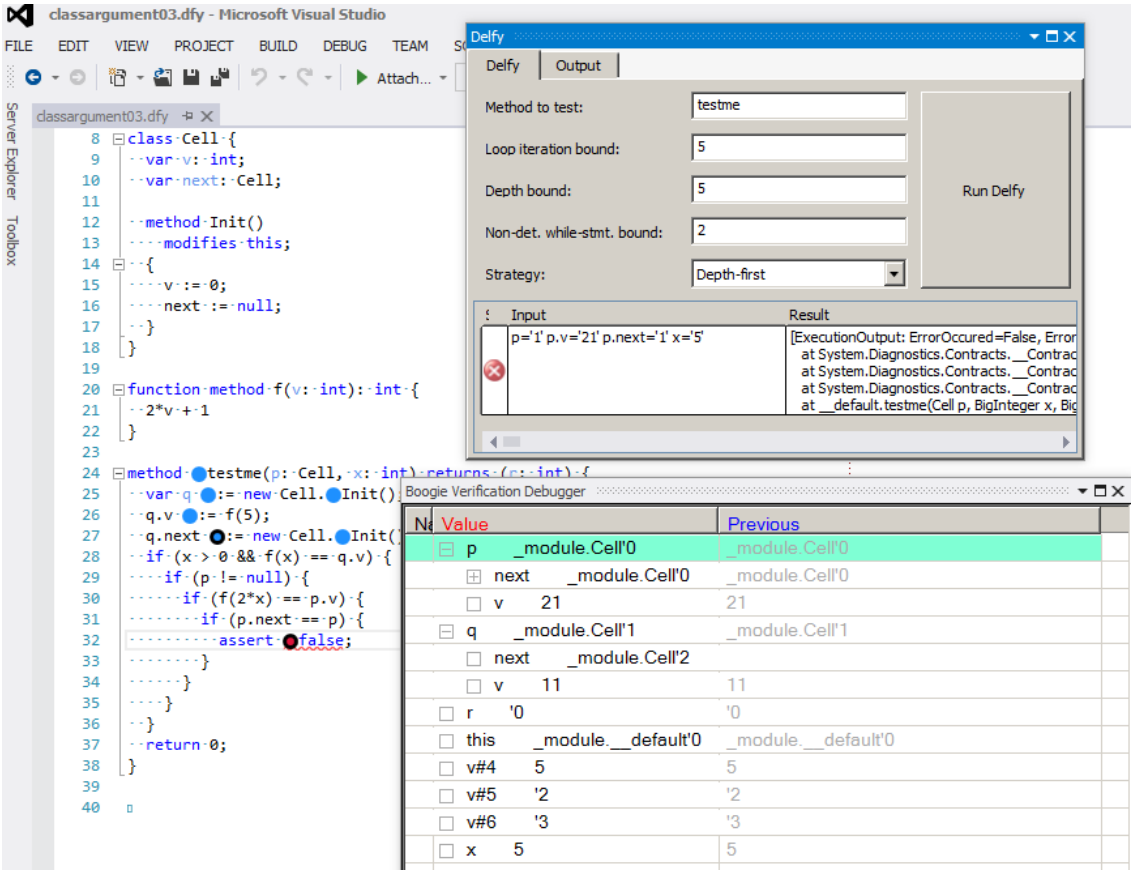

Figure 6.1: BVD-Delfy comparison for the case of object graphs.

would be needed in order to trace back the values. This is not the case. As indicated with the blue dot before the while statement (Figure 6.3), BVD provides only the states before the while loop and at the assertion.

Sets and sequences

For the case of sets and sequences we observed that if Delfy is able to come up with concrete inputs, BVD is as well. Figure 6.4 shows a very simple example of this.

BVD handles sets and sequences by axiomatizing them at the level of Boogie. Therefore, they are directly encoded in the verification conditions. For this reason the theorem prover is able to create accurate models for those.

We conclude that the goals of Delfy and BVD are different. Delfy is built to generate inputs for the unit under test. BVD in contrast is a debugger that means that its goal is to be a general tool to understand verification errors. This means both tools complement each other in helping to find the reasons for the verification errors that Dafny reports.

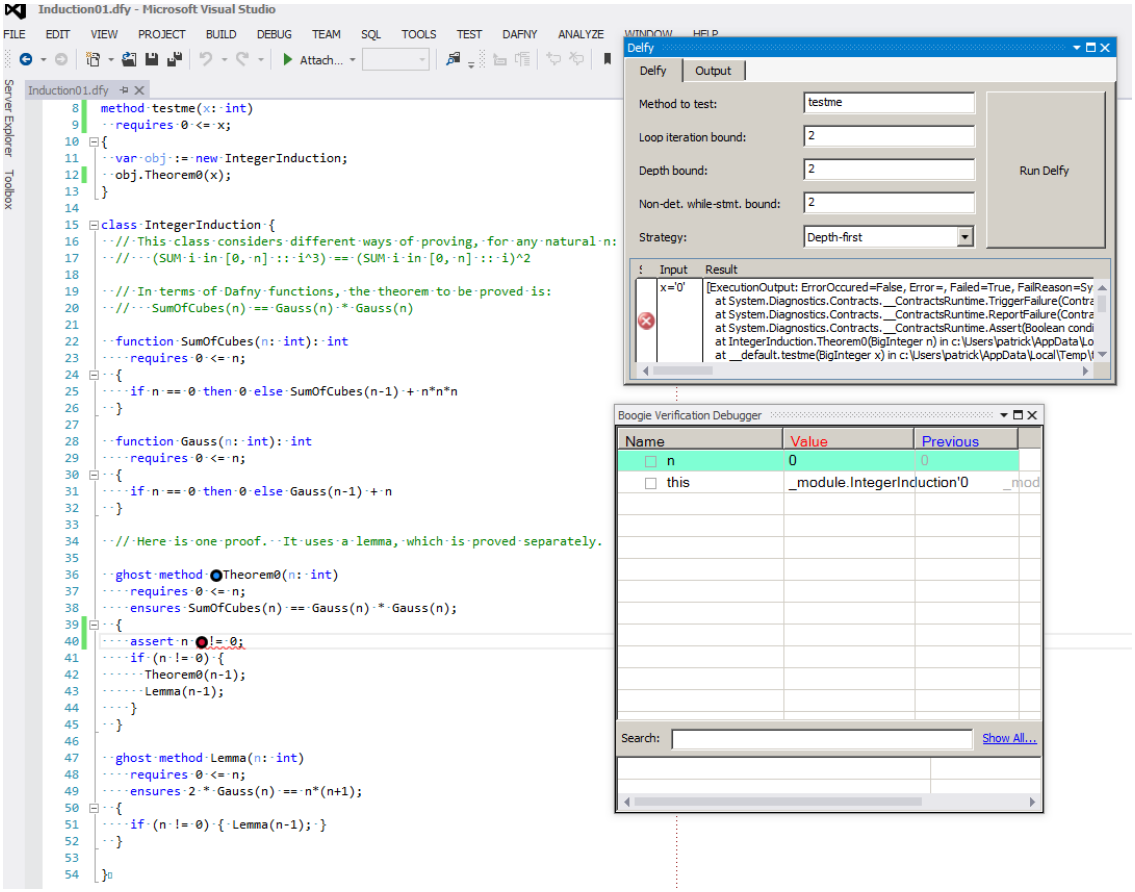

Figure 6.2: BVD-Delfy comparison for the case of method calls.

6.2.4 Comparison with Pex

We implemented support for various features of Dafny, which are specific to verification. For example there is support for pre- and postconditions, termination metrics and loop invariants. These are examples that can be handled by Pex as well, assuming the C# code is properly annotated using Code Contracts. But in addition we also implemented support for sets, sequences, modifies clauses, reads clauses and fresh expressions in Delfy. In order to support those features we need to keep track of changes, which is not supported by Code Contracts and Pex. In comparison to Delfy, Pex is also different in that Pex works at the level of byte code whereas Delfy works at the level of the Dafny AST.

In this Section we describe the experiments we did in this regard and present the results we obtained.

We conducted the experiments in the following way. We took a method to test from the test suite. We then created three different versions of this test. The first one verifies. For the other two tests we introduced different errors. We then ran Delfy and Pex and gathered the results. Table 6.3 summarizes the results of the experiments. The first column contains the name of the test case. The second column the description of the different versions of this test case and the other two columns the results when running Delfy and Pex on these cases.

The results of the experiments show that Pex is able to deal with modifies and reads clauses. The reason for this is that we generate the needed code during compilation of a Dafny program to C#.

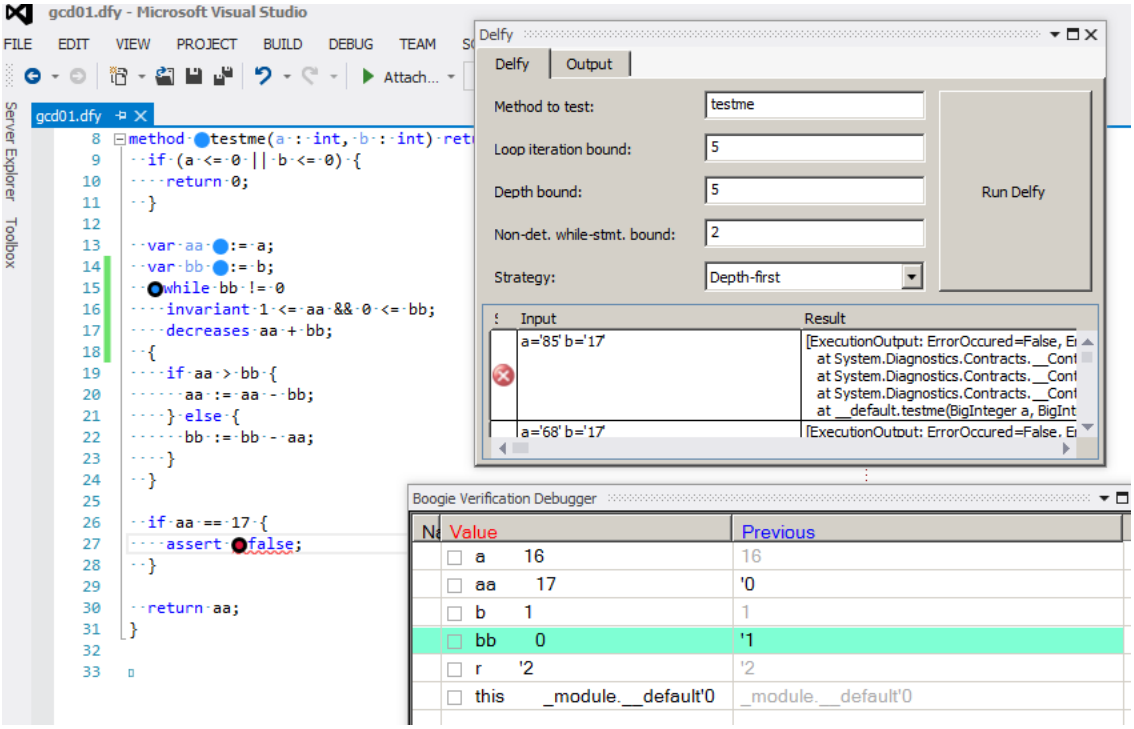

Figure 6.3: BVD-Delfy comparison for the case of loops.

Delfy does nothing special in the symbolic execution in addition to these runtime checks during the concrete execution. This is the reason for Pex being able to deal with those constructs.

However, Pex cannot handle non-determinism, sets and sequences. In particular Pex cannot deal with non-deterministic if statements, while loops and assignment statements. The reason for this is that the Dafny-to-C# compiler does not generate C# code for handling those. For instance, a non-deterministic if statement is translated by just compiling the code of one of the branches. The reason why Pex cannot deal with sets and sequences is that they are compiled using C# reference types. Therefore, Pex for example generates `null` as a possible input and it cannot make use of Dafny's helper functions for sets and sequences. It also violates the precondition if the precondition contains expressions with sets and sequences. Listings 6.1, 6.2 and 6.3 show examples of Dafny test cases with which we observed this behavior of Pex.

In the generated C# code, Dafny helper classes for sets and sequences are used. For the case of sets we also manually changed the generated code to use .NET `HashSets`. Pex cannot deal with sets even when using `HashSet`. The results are similar to those when using the helper set class.

```
method testme(s: set<int>) {
  assert {1} == s;
}
```

Listing 6.1: set02.dfy

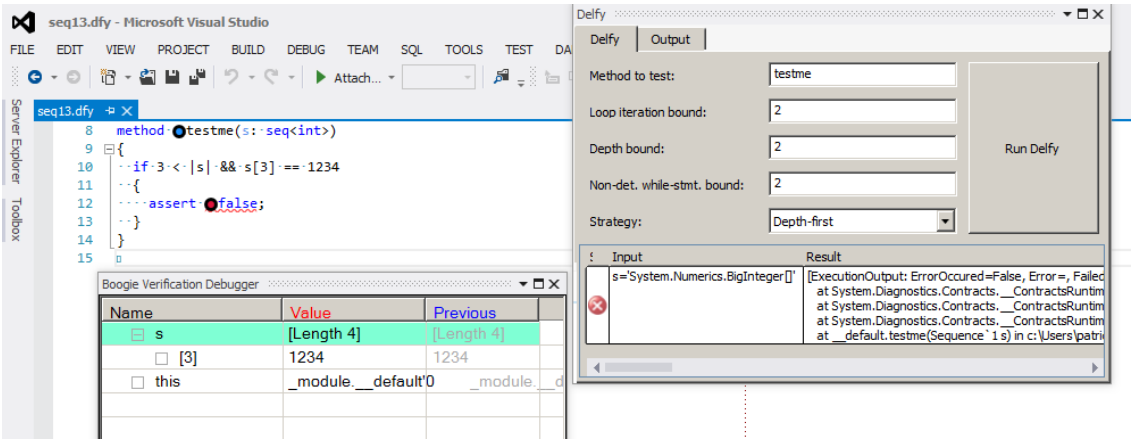

Figure 6.4: BVD-Delfy comparison for a simple test containing a sequence.

```
method testme(s: seq<int>)
  requires 4 in s;
{
  if 1 in s
  {
    if 2 in s
    {
      if 3 in s
      {
        assert false;
      }
    }
  }
}
```

Listing 6.2: seq02.dfy

```
method testme()
{
  var x := *;
  assert x == 0;
}
```

Listing 6.3: nondeterministicassignstmt01.dfy

Test case	Changes	Delfy	Pex
`AdvancedLHS.dfy`	verified	✓	✓
	introduced one error in modifies clause	✓	✓
	introduced an additional assertion error	✓	✓
`set02.dfy`	no changes, one assertion that fails	✓	✗
`seq02.dfy`	verified	✓	✗
`nondeterministicassignstmt01.dfy`	verified	✓	✗

Table 6.3: Results of comparing Delfy and Pex. Every line corresponds to one experiment. The first column states the name of the test case. The second column explains the test, i. e., whether it verifies or the changes we made. The other two columns state the results: ✓ means expected result (i. e., expected error found or no error found), ✗ means unexpected result (i. e., error not found or unexpected error found).

6.3 Limitations

6.3.1 Incompleteness

Delfy's dynamic test generation engine does not explore the whole state space. In particular there are configurable bounds on the number of loop iterations, recursion unfoldings and on the depth of the search tree. If a bound is exceeded this is reported and the corresponding path is not further explored. Furthermore, if Z3 is not able to solve a given condition, the exploration of the corresponding path is stopped (timeout).

During the evaluation phase of the project we observed that setting these configurable bounds appropriately depends on the test case. As an example, for a method that computes the n-th Fibonacci number recursively in exponential runtime, the bound on the number of recursion unfoldings must be chosen small enough. But this then limits the number of possible errors that can be found. Thus, for another test case that does not involve recursion, this bound can be set higher. But then perhaps the bound on the number of loop iterations must be adapted if it contains loops.

6.3.2 Unsupported Dafny features

The following list enumerates the main Dafny features that are not supported by Delfy.

- `DatatypeDecl` (algebraic data types)
- `IteratorDecl` (iterators)
- Unbounded `AssignSuchThatStmts`
- `CalcStmt`
- `TernaryExpr`
- Non-exact `LetExprs`
- Non-deterministic assignment statements are only supported for `int`, `nat`, `bool` and reference types
- Functions with no body are only supported for a return value of `int`, `nat` and `bool`
- Rank comparisons
- Decreases clauses: only `int` and `nat` type supported
- Sets: only membership, equality, inequality, union, intersection, difference operations are supported
- Sequences: only length, membership, equality, inequality operations are supported
- Maps
- Arrays

Chapter 7

Conclusions

We have presented and implemented Delfy, a dynamic test generation tool for Dafny. We applied known techniques of dynamic test generation to a language specially designed with verification in mind. Therefore, we had to go beyond the known approaches and we had to come up with new techniques to support certain features that are special to a verification language like Dafny. Examples include the compilation to actual code, the support of ghost features, the support of non-determinism and the support for the special value types in Dafny (sets, sequences, maps).

Dafny is a language with static verification in mind. Thus, there is in principle no need to compile it to actual code that can be run. But in order to apply software testing methods, in particular dynamic test generation in our case, we need to actually run it. Dafny already contained a compiler for compiling Dafny to C# code. However, we had to extend it in order to fit it to our needs of dynamic test generation. Ghost state is another example of special feature of a language with static verification in mind. To support this we had to come up with a lot of extensions to the existing Dafny-to-C# compiler. In addition we had to deal with non-determinism. We handle this similarly to model checker tools. This means that we explore all the possibilities of a non-deterministic choice. Another challenge was to add the support of the set type. For this we had to research ideas. Furthermore, we discussed this issue and got valuable ideas on how we could approach it.

We have enhanced the Dafny Visual Studio integration to support Delfy. This means that there is a graphical front end available for Delfy. For this we also added a static symbolic executor to Delfy. This enables the support of selecting a Dafny verification error in Visual Studio and then generating only inputs that cover the error using Delfy.

In addition we have also evaluated Delfy. We have created a test suite of Dafny programs. This test suite was used to see how many programs Delfy supports. In particular we pointed out which features of a static verification language are difficult to handle in dynamic test generation in general. Furthermore, we showed the effects of a static analysis prior to the actual dynamic test generation. We also compared Delfy to the Boogie Verification Debugger. By doing this we were able to show that Delfy complements the Dafny verifier and BVD well. Moreover, we compared Delfy to Pex for verification features that Delfy supports but not Pex.

7.1 Related work

A dynamic test generation tool for .NET is Pex [19]. Pex is a white-box test generation tool based on dynamic symbolic execution. It works at the level of CIL byte code and therefore supports safe

.NET programs. Moreover, it also supports a commonly used set of unsafe features of .NET such as memory accesses involving pointer arithmetic. In addition, it is able to deal with particular verification constructs if the code under test is annotated using Code Contracts [16]. Pex generates test inputs for parameterized unit tests, i. e., for a method under test, with the goal of achieving high statement coverage. A parameterized unit test is a method that takes parameters, exercises the code under test and then asserts properties of the expected behavior. It employs search strategies that are tailored to achieve high branch coverage in a short amount of time.

For C code there is CUTE [18]. There is also a version of this tool for Java. It is based on the approach of separated concrete and symbolic execution. This means it concretely executes the unit under test to generate a trace. This trace is then used for symbolic execution. In comparison to other dynamic test generation tools, the main difference of CUTE is that it aims to support units under test that take memory graphs as inputs. For this it keeps a logical input map that represents all inputs as symbolic variables. Integer constraints and pointer constraints are treated separately. Pointer constraints are also simplified using the logical input map in order to make the process of constraint solving more efficient.

Another tool that supports C code is DART [6]. DART introduced concepts such that unit testing can be done completely automatically on any program that compiles. It does this in three steps. First, the interface is automatically extracted using a static analysis approach. For this interface a test driver is then generated as a second step. This driver performs random testing to simulate the external environment. Then, the third step is to perform dynamic symbolic execution to automatically generate new test inputs to direct the execution along alternative program paths.

SAGE [7] is a whole-program white-box fuzz testing tool that works on the level of x86 code. It is based on instruction-level tracing and supports system testing of arbitrary file-reading Windows applications. SAGE is thus a tool for system testing, i. e., it not only supports testing single units but whole programs as well. It is based on dynamic symbolic execution and is therefore capable of finding bugs that are beyond the reach of black-box testing tools. SAGE introduced the concept of the generational search strategy. Being a whole-program testing tool, the generational search strategy helps in that it systematically but partially explores the state space of large programs.

7.2 Future work

There are different directions for future work. For instance adding support for more Dafny verification features would enable Delfy to support more programs. Another example of future work is to make better use of the Dafny verifier's results.

Many features that are crucial for a verification language cannot be handled by Delfy. For this reason the tool cannot be used for more interesting verification examples and challenges. We think that going into this direction would be an interesting challenge.

Our approach to make use of the Dafny verifier's results is very conservative, i. e., we prune a lot. Improving this is an interesting challenge as well.

Bibliography

[1] 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*, LNCS, page 364387. Springer, 2006. 1

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *SMT*, 2010. 43, 44, 45

[3] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *TPSV*, volume 7682 of *LNCS*, pages 1–30. Springer, January 2012. 48

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. 1

[5] Leonardo De Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52. IEEE, 2009. 42

[6] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005. 3, 68

[7] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008. 3, 11, 12, 13, 15, 68

[8] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568. Springer, 2003. 20

[9] James C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385394, July 1976. 2

[10] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The Boogie Verification Debugger. In *SEFM*, pages 407–414. Springer, 2011. 60

[11] K. Rustan M. Leino. This is Boogie 2, June 2008. 1

[12] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010. 1

[13] K. Rustan M. Leino. Dafny project web page. `https://dafny.codeplex.com/`, 2013. [Online; accessed 04-September-2013]. 1, 17

[14] K. Rustan M. Leino. Getting Started with Dafny: A Guide. `http://rise4fun.com/Dafny/tutorial/guide/`, 2013. [Online; accessed 04-September-2013]. 1

[15] Microsoft. What Is Windows Communication Foundation. `http://msdn.microsoft.com/en-us/library/ms731082.aspx`, 2012. [Online; accessed 07-November-2013]. 7

[16] Microsoft Research. Code Contracts. https://research.microsoft.com/en-us/projects/contracts/, 2013. [Online; accessed 20-September-2013]. 7, 68

[17] Microsoft Research. Getting Started with Z3: A Guide. http://rise4fun.com/z3/tutorial/, 2013. [Online; accessed 28-September-2013]. 42

[18] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC*, pages 263–272. ACM, 2005. 3, 68

[19] Nikolai Tillmann and Jonathan de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008. 67

[20] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Precise identification of problems for structural test generation. In *ICSE*, pages 611–620. ACM, 2011. 20