Integrating dynamic test generation with sound verification

Patrick Emmisberger

Research in Computer Science

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

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

07/02/2015

Supervised by:
Maria Christakis
Prof. Dr. Peter Müller

Abstract

In this project we set out to find ways to combine sound verification with dynamic symbolic execution in the form of a wizard that aggregates the results from both tools and provides helpful feedback to the developer. We use Dafny, a programming language focused on sound static verification and Delfy, a dynamic symbolic execution engine for Dafny, with the aim to build an extension for the preferred development environment for Dafny: Microsoft Visual Studio. We define a new program abstraction that is easy to analyse statically and run dynamic symbolic execution on, yet is still powerful enough to model the semantics of most programming languages, in particular of Dafny. Based on this abstraction, we perform a static and dynamic analysis and augment this information with the results of the existing sound verification for Dafny. Finally, we present how this information is used to insert annotations and hints into the source code to support the developer.

Chapter 1 introduces the technologies and previous work on which this project is based. Chapter 2 continues by presenting the Delfy Engine, a framework that implements dynamic symbolic execution in a language-agnostic way based on a model of the source program. Subsequently, Chapter 3 shows how Delfy was adapted for Dafny specifically. Chapter 4 concludes with a description of the feedback that we can provide the developer with, as well as how this feedback is integrated into Visual Studio and the existing IDE tooling for Dafny.

Contents

Chapter 1

Introduction

In this project we explore the possibilities that arise from combining sound verification with dynamic symbolic execution. In particular, we want to build an extension for Microsoft Visual Studio (from now on referred to as “wizard”) that simplifies working with Dafny by providing direct feedback on the source code currently under development. In the following sections we give an overview of the key technologies and ideas which this project is built on. This chapter concludes with an introduction of the concrete features and type of feedback that we want to provide as part of the wizard and our motivation for the main decisions that were made during the project.

1.1 Dafny

Dafny[13] is a programming language with built-in specification constructs to support sound static verification. For verification, the Dafny program is first compiled into IVL (*Intermediate Verification Language*). This intermediate representation is passed on to the Boogie verifier[1], which internally relies on the Z3 constraint solver[5]. Dafny has a class-based type system and provides language constructs for specifying pre- and postconditions as well as loop invariants among others. The specification constructs are complemented with special syntax for sets and sequences as well as advanced features like non-deterministic statements and uninterpreted functions. The Dafny compiler produces C# code but omits all specifications which are used only during verification.

An extension[14, 15] for Microsoft Visual Studio provides language support, debugging tools[12] and automatic background verification while editing. Verification errors are displayed directly at the respective locations inside the editor, allowing the user to quickly respond to problems with the ultimate target of finding a proof of functional correctness.

1.2 Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) is a white-box test generation technique[7]. Its roots lie in an approach called “fuzzing” where the inputs of a program (e.g. a file or the parameters of a function) are modified randomly or using heuristics to provoke software errors[10, 2, 19]. This is also called “black-box fuzzing” since it does not take the code of the program being tested into account. Dynamic symbolic execution uses the same idea but takes a different approach to generating input values. The code is executed with a set of concrete values, but simultaneously these values and their usage are tracked symbolically, indicating to the testing tool how to change the values to provoke different execution paths in the code under test.

As a concrete example consider Listing 1.1. Assume the first execution uses parameter values `a = 0; b = 0;`. The test `a > 10` will return `false` and the method returns. The symbolic execution engine observes all branch conditions and generates a *path condition* that constrains the input values, so that for all input values that satisfy the constraint the same execution path will be taken. For our concrete example the path condition for the first run is $a \leq 10$.

```
void Test(int a, int b){
  if (a > 10) {
    if (b > a*2)
      throw new Exception("Unexpected
          Error");
  }
}
```

Listing 1.1: Example code for DSE

In order to explore more parts of the method, the path condition is modified (normally the last conjunct is inverted) and by using a constraint solver, new values for the inputs are found. In our example the new path condition would be $a > 10$ and we assume that the constraint solver returns `a = 11`. `b` is not modified and still uses its default value of 0. For the next run, the first test holds and the *then branch* is executed, containing a second condition. The second condition will evaluate to `false` for this run and the method returns. The symbolic execution engine provides the new path condition $a > 10 \wedge b \leq a * 2$. We invert the last part of the path condition resulting in $a > 10 \wedge b > a * 2$, for which the constraint solver returns the values `a = 11; b = 23;`. We run the method with this input and the exception is thrown.

Using this technique, the testing tool tries to maximize branch coverage and find input values for which the method behaves unexpectedly.

Listing 1.2 contains the basic algorithm in pseudo-code. `solve` runs the constraint solver to find concrete values that satisfy the given condition. `execute` runs the method under test with the given input values and returns the path condition that was generated in this run.

```
void explore(IEnumerable<Term> pathCondition) {
  values = solve(pathCondition);
  if (pathCondition is satisfiable) {
    newPathCondition = execute(values);
    for(int i = |pathCondition|; i <= |newPathCondition|; i++) {
      var prefix = newPathCondition[1..i];
      prefix[i] = ¬prefix[i];
      explore(prefix);
    }
  }
}
```

Listing 1.2: Dynamic Symbolic Execution Algorithm (Pseudo-Code)

1.3 Delfy

Delfy implements dynamic symbolic execution for Dafny and is the result of Patrick Spettel's master thesis[17]. The existing Dafny to C# compiler was extended to emit Code Contracts[8] annotations for checking the specifications at runtime. To track the execution symbolically, the compiler was modified to emit instrumentation code along with the output program that provides different types of callbacks at runtime. The callbacks are mainly used to send code snippets of the executed Dafny code back to the DSE engine and can be viewed as an execution trace on the source level. The engine parses the code snippets and executes the parsed code symbolically.

Delfy supports different exploration strategies as well as in-process and out-of-process exploration. The later allows the DSE engine and the program under test to run in different processes. This is desirable because the DSE engine is usually tightly coupled with the development environment and the exploration procedure could interfere with the operation of the IDE. To implement out-of-process exploration, Delfy relies on the Windows Communication Foundation framework to marshal the callbacks between processes.

In this section we summarised the approach of the already existing version of Delfy. For this project we heavily modified Delfy to provide the features motivated by the next paragraphs.

1.4 Goals and Motivations

Through the combination of static verification and dynamic test generation we want to achieve higher developer productivity by reducing the effort of debugging spurious verification errors and also lower the need for specifications. To achieve this goal, we propose the following features:

- Run the verifier and DSE in parallel and combine the results of both tools. If the verifier can verify an assertion, it is not necessary to attempt to generate a failing test case for it. Likewise, if the verification cannot prove an assertion but DSE has full coverage without an assertion violation, we definitely found a spurious error.
- Prioritise assertions based on the (path) coverage that can be achieved by DSE. The lower the coverage, the less information DSE can provide, neither for verification nor in the form of counterexamples.
- Strengthen the verification by converting assertions into assumptions if they can be proven by full exploration of straight-line code. Because DSE is less modular than static verification, it does not rely on summarisation using postconditions and invariants, thus reducing the need for specifications in cases where DSE is sound.

As we set out to implement these features, we soon realised that DSE alone does not provide the necessary information for proving properties of straight-line code. We also needed a static analysis that calculates the control flow graph (CFG) and can provide an upper bound on the number of paths through a method. This confronted us with two main problems:

Delfy tracks the symbolic values by sending a source level trace of the code being executed back to the DSE engine. However, the code snippets cannot be easily mapped to elements of the AST, making it difficult to combine the callback information with the results of the static analysis.

Dafny uses an internal representation (IR) that is relatively close to the original parse tree. This makes it difficult to build proper control flow graphs because Dafny supports various types of loops and control flow structures, has short-circuit evaluation of logical “and” and “or” operations and provides a conditional expression similar to the C-style ?: ternary operator. This would require the CFG to describe transitions between partial expressions, resulting in a CFG that is very difficult to work with.

These problems in combination with other unsatisfactory circumstances, like the performance of the out-of-process exploration as implemented by Delfy, lead us to the conclusion that we need another abstraction layer that has simpler semantics than the Dafny AST and that the callbacks should be tightly coupled to this abstraction. It became clear that the existing Delfy code had to be fundamentally changed, so that it can be used as a base for implementing the features described previously.

Chapter 2

The Delfy Engine

2.1 Introduction

In this chapter, we introduce the new abstraction layer that was previously motivated in Section 1.4. We designed the abstraction in a language-agnostic way, such that it completely separates the static and dynamic analysis from the source language. The intermediate representation was chosen to consist only of a few distinct node types and to be easy to analyse, while still being powerful enough to represent all features found in the compilable subset of Dafny or any mainstream language. This not only allows us to solve the problems posed by this project in a (compared to a real language) simple representation, but also to easily transfer the results to source languages other than Dafny.

This fundamental change in the architecture of Delfy resulted in a complete rewrite. For the remaining part of this report we refer to the initial implementation as the *original Delfy*, while *Delfy Engine* or just *Delfy* describes the new implementation.

This chapter introduces the intermediate representation used by the Delfy Engine as well as its internal structure and extensibility points.

2.2 Program Model (Intermediate Representation)

Delfy abstracts from a concrete source language by working on a *program model.* A program model describes the types and methods of a program as well as their precise operational semantics. To use the Delfy Engine for a particular source language, an adapter must be provided that can at least build the program model from a program in the source language. If the adapter is bidirectional (i.e. it can map elements of the program model back to the original source language elements), the Delfy Engine can provide feedback directly in terms of the source language. It provides its own type system, instruction set and is accompanied by a type checker and compiler.

Delfy represents code as *Instructions* and *Terms.* Instructions may have side-effects and contain control flow. Terms represent the building blocks of values (constant values, local variables, etc.) as well as a way of combining them into more complex expressions (operators, field access, array indexing etc.). The evaluation of a term may not have any side effects.

The following sections describe Delfy's model for representing types, methods, statements and expressions.

2.2.1 Type System

The Delfy program model can be seen as a statically typed programming language, meaning that every term is assigned a single type before execution. During execution, every value that results from the evaluation of a term must have a type that is compatible to the term's static type. By default, the notion of compatibility is defined over the subtype relationship. A type T is compatible to another type U, if and only if the types are equal or T is a subtype of U. However, this behaviour can be replaced. Finally, the static type determines which fields, methods and operators are supported by a value of said type.

The Delfy type system is object oriented and comparable to the type systems found in mainstream languages like Java or C#. However, it deliberately does not define the concrete semantics of types like equality, compatibility etc. so that it may be adapted to fit the type system of the source language.

Delfy differentiates between two different kinds of types. Nominal types are identified by a name. Type references, on the other hand, have no name and are usually modifiers or combinations of other types. The only kind of type reference currently implemented are *generic type instantiations*, but examples for other types in this category would be tuples, pointers and C++-style references.

Nominal types provide a namespace and a name. The concatenation of both is called the *fully qualified name* and must be unique in a program. Namespaces are a way of grouping types and allow to avoid name conflicts. This feature exists only for developer convenience, as for the Delfy Engine namespaces have no semantic meaning attached and types are always referred to by their fully qualified name. Every type can provide fields and methods. Because the Delfy IR is not used as a "language" that a programmer uses directly, it does not support visibility modifiers.

Methods have a name and can have zero or more parameters and results. The name of a method does not have to be unique as long as it is possible to uniquely identify the receiver method at all call sites (e.g. by using the parameter types, i.e. method overloading). The default method binder uses the name and the types of all parameters to identify a method. However, this behaviour can be replaced to match the source language.

Operators are implemented as methods with special names. When an operator is used, the type of each operand is searched for a matching operator method. The search is conducted from left to right and stops as soon as a matching method is found.

Methods and fields can be attached to an object instance or be *static*. Static fields and methods do not expect a receiver object when they are referenced. The value of a static field is shared across all instances of the type and can be thought of as a global variable.

The Delfy type system supports generics. Nominal types and methods can have generic type parameters. In case a type has at least one generic parameter, we refer to it as a *generic type definition* or an *open type*. Generic type definitions cannot be used directly in a program, but must first be instantiated by providing a *closed type* for each generic parameter. When a type has no generic parameters or was fully instantiated it becomes a closed type. Similarly, a *generic method definition* must be instantiated at every call site.

Only the most basic types are directly built into Delfy. The reason for this is that many languages have subtle differences in their type systems. The more built-in functionality is provided, the greater are the chances that some aspect of an implementation is incompatible with a source language, thus preventing the usage of Delfy in combination with said language. The following types are directly built into the Delfy engine:

- **Null.** The null type only has a single value which refers to a non-existing object (the *null* *literal*). The null type takes a special role, because it is assignable to all reference types.

- **Object.** The root of all reference types. The object type only provides the equality operator that uses its identity for equality.
- **Boolean.** The boolean type consists of two values: true and false. Each element of the path condition as well as the path condition itself is of this type.
- **Invalid.** The invalid type is used for terms that cannot be properly typed. It has no value and does not support any conversions. The Delfy IR requires all terms to have a type assigned to them. However, there are cases where this is not possible (e.g. an operator is applied to arguments that do not support this operator). In that case the term is typed as *invalid.* This is detected by the type checker and a proper error message is generated.

The user can define additional nominal types, kinds of type references, override the method binding logic and replace type compatibility and equality.

2.2.2 Terms

Terms are versatile building blocks for representing values. A term is either a leaf-node (e.g. literal value, local variable, symbol) or a composite node (operator, field access, indexer) built from other terms. Every term must have a single type, which also can be *invalid.* Terms have multiple purposes, as they represent code that results in a value (e.g. the right hand side of an evaluation or the condition of a control flow instruction), are used to represent the clauses of the path condition and are also used as containers for values that are passed to the user program.

A central problem is the conversion between different types of representations of a value. There are the AST representation of the source language, the AST representation of the constraint solver, the runtime values that are passed to the concrete execution of the program and the in-memory representation of these values, between which we need to convert. This problem is solved using a concept called *term constructors and destructors.* A term constructor is a transformation that converts a value into a term (e.g. from the constraint solver AST to term representation), a term destructor is the inverse transformation. To perform a concrete execution using a value generated by the constraint solver, we first use a term constructor to create the term representation from the model provided by the constraint solver and then deconstruct the term into a runtime value that can be used by the program.

Terms are serialisable, which allows to persist terms or exchange them between processes. This property is essential for rerunning explorations and perform out of process exploration as described in Section 2.4.6.

2.2.3 Instructions

The instruction set of Delfy has been kept small to make it easier to analyse. The following instructions are supported:

- **Assignment.** Evaluates a source term and assigns its value to a target term.
- **Branch.** Evaluates a condition and, if it evaluates to *true*, performs a jump to an arbitrary instruction in the same method. If the condition is *false*, the next instruction is executed.
- **Call.** Calls a method with a set of argument values and stores the results in the given target terms.
- **Constraint.** Asserts or assumes that a specific condition holds.

- **Return.** Exits from the current method.

Even though the instruction set is very simple, it is powerful enough to represent arbitrary looping constructs, recursion and side-effects.

2.3 Static Analysis

Based on the program model, the Delfy Engine performs a number of static analyses that are used internally, but are also available to clients to gain a more detailed understanding of the source program.

As a first step, the control flow graph is calculated. The instructions of a method are first grouped into uninterruptible segments called *basic blocks*. The basic blocks are then connected by transitions that model the control flow. Transitions can have a condition that must hold, if that transition is to be taken. However, at the end of each basic block, at least one transition must be navigable. This allows us to calculate the number of paths through a method, detects loops and reveals dead code.

From the control flow graph, assignments to local variables are versioned to derive the static single assignment (SSA) form of that method. The SSA representation is usually found in the optimiser pass of a compiler, but in our case we use it to provide information about data flow and uninitialised variables.

By combining the data flow information with the transition conditions between basic blocks, we can under-approximate an invariant for each entry and exit state of a basic block. Using these invariants, we can prune paths and unfeasible counterexamples.

2.4 Dynamic Analysis

2.4.1 Exploration Engine

The following sections provide an overview of the internal structure and introduce the terminology of the exploration engine.

An *exploration* takes as input a program model and a method runner that allows to execute the program concretely with enabled instrumentation. The method runner is either provided for a specific source language or obtained by compiling the program model into executable code (see Section 2.4.5 for more information).

A single execution is called an *exploration run*. During an exploration run, the program is both executed concretely and symbolically. The parameters for an exploration run are the method to explore, a path condition and a set of initial symbols (e.g. for parameters or heap structures). As a first step, the constraint solver tries to find values for all symbols such that they satisfy the path condition. If the constraint solver fails to generate values (either because of a timeout or because the path condition is unsatisfiable), the exploration run ends with the respective error status. Otherwise, the program is executed concretely with the generated values while the data and control flow is tracked symbolically. If an assumption is not met, the path condition is extended by the assumptions's condition, the execution is aborted and no error is generated. If an assertion is violated, we also append it to the path condition and we abort with an error. If any other exception (e.g. null reference exception) occurs, we also abort with an error. All the information of an exploration run (parameters, initial input values generated by the constraint solver, concrete and symbolic output values, statement and branch coverage, the final path condition and the error information) is combined into an *exploration node*.

Independently of an error, we pass the exploration node to a *successor provider* that generates a number of parameters for additional exploration runs. The default implementation uses the path expansion described in Section 1.2. If there is no existing exploration node with exactly the same parameters, we add the parameters to the *search frontier*. The search frontier contains all parameter sets for which no exploration run has been executed yet. The default implementation uses a breadth-first search to traverse the search frontier but this behaviour can be replaced to use a depth-first, generational or custom algorithm instead.

When the exploration either reaches one of the limits (maximum duration, node count, run count) or the search frontier is empty, the exploration ends and all exploration nodes are stored inside an *exploration report*. The nodes are stored as a directed graph, where an edge represents the successor relationship between two nodes.

2.4.2 Constraint Solving

Delfy uses the Z3[5] constraint solver by default. Z3 is a theorem prover built by Microsoft Research which is released under the MIT license. Delfy includes term con-/destructors compatible with Z3 for its built-in types. For custom types, the users can define their own mapping, thus allowing to implement e.g. set types or arrays directly using the Z3 array theory[6].

Z3 uses value-based equality and has no concept of object identity or subtyping built-in. If we want to allow type checks or object equality based on identity as part of the path condition, we need to map these concepts to structures that Z3 can reason about. The following approach was chosen to map subtyping, polymorphism, nullability and object identities to Z3 using integers as well as array and set theory.

We first translate the type hierarchy in a program to sets of integers. Let T be the set of all types in our program, $Sub(t)$ the set of all direct subtypes of t and $Id(t)$ a function that maps a type $t \in T$ to a unique natural number, s.t. $\forall t, u \in T. \quad t = u \Leftrightarrow Id(t) = Id(u)$. For each type t we define Z3 constant τ_t that represents the type t. τ_t is defined as an integer set $\tau_t = \{Id(t)\} \cup \bigcup_{u \in Sub(t)} \tau_u$. Using τ we can now express the subtyping relationship $\rhd$ in Z3 using $t \rhd u \Leftrightarrow \tau_t \supset \tau_u$.

Next, we transfer object identities to integers. For each symbol o that represents an object instance we introduce a Z3 variable of type int ω_o. Object equality simply translates to integer equality, s.t. $o = p \Leftrightarrow \omega_o = \omega_p$. We reserve 0 as a special object identity for *null*, thus nullity translates to $o = null \Leftrightarrow \omega_o = 0$.

Now we want to impose a typing restriction on instance o. For that we define a Z3 array $\pi : \mathrm{I} \rightarrow \mathrm{N}_0$ that represents the type of object o. In Z3 types this is simply a mapping from integer to integer. To represent $t \rhd typeof(o)$, we assert $\omega_o \neq 0 \Rightarrow \pi(\omega_o) \in \tau_t$. We need the presumption $\omega_o \neq 0$ because *null* is a valid value for all reference types. Z3 must now choose $\pi(\omega_o)$ as an element of τ_t, which is exactly the set of all $Id(u)$ for which u is a subtype of t (particularly u can be equal to t). If it cannot find a type, the expression is unsatisfiable.

To translate restrictions over fields like $o.f = x$, we create an array for each field of $typeof(o)$. Similar to π for the type of an object, we translate the former constraint to $\omega_o \neq 0 \wedge f(\omega_o) = x$. We require the non-nullity condition to prevent setting fields on *null*. This allows to prove assertions like

$$o.f \neq o.p \Rightarrow f(\omega_o) \neq f(\omega_p) \Rightarrow o \neq p. \tag{2.1}$$

This translation of object identity is a big departure from the translation used in the original version of Delfy, since it did not take into account subtyping (objects of different types are always different) and aliasing of fields (violation of (2.1) possible).

The translation is implemented as described above with the exception of the definition of set T. The problem is that for real programs, creating the structure τ for all types would result in a large overhead and in the presence of generics, this set becomes infinitely large. For that reason we only include types that are required for expressing the constraint that we want to translate. However, this can result in unsatisfiable constraints when we have type constraints using generic types or multiple subtyping, because even if a type exists that would satisfy the path condition, it is potentially not included in T.

2.4.3 Exploration Extensions

In many cases it is desirable to influence the exploration behaviour, e.g. to guide the exploration to certain paths, handle loops in different ways or provide support for language constructs that are not supported out of the box. Exploration extensions provide a modular way to replace or extend the default behaviour by either implementing one of the extensibility interfaces (e.g. for changing the search strategy), providing additional successors to an exploration node or directly hooking into the symbolic interpretation.

The set of active exploration extensions can vary between each exploration, providing a flexible way of exploring the source program using different strategies.

2.4.4 Handling input-dependent loops

One application of an exploration extension that is included in the Delfy Engine is an alternative way to treat input-dependent loops. Input-dependent loops have always been problematic for testing because full coverage can never be reached through unrolling. However, if loops are annotated with an invariant, we can apply a different strategy than simple unrolling.

The primary idea is to view the loop body as an independent method, using the loop invariant as a pre- and postcondition. Furthermore, the precondition is strengthened with the loop guard. The loop targets (i.e. the values that are modified during the loop execution) are modelled as parameters. The loop iterations can now be viewed as individual method calls. If the loop invariant is too weak, this is of course an over-approximation that leads to false positives.

We implemented the aforementioned approach using a combination of static analysis and extending the symbolic interpreter. By hooking into the execution callbacks, we check on entering a loop, if the symbolic loop condition contains an input symbol (in which case the condition is input-dependent). If this holds, we havoc all loop targets, assume the loop guard and execute the loop body. The first instruction inside a loop is the assumption of the invariant, constraining the new symbols for the loop targets to the loop invariant. Then we continue exploring normally until we reach the end of the loop. When exiting the loop, we havoc the loop targets again for the remaining part of the method and assume the negation of the loop guard. Havocking loop targets is only problematic if they are aliased object references. In the presence of aliasing, the potential aliases must be tracked and also be havocked.

2.4.5 Execution Models

Delfy supports two different execution models. The best choice for the execution model depends on the source language and its runtime environment. In summary, the first execution model only tracks the execution using a callback on every IR instruction. The second execution model compiles the IR and adds the callbacks during compilation automatically.

Callback-based Monitoring

In this execution model, the instrumentation of the source program obliges the adapter for the source language. Delfy only relies on a small set of callbacks to track the symbolic state of the program. The callbacks that are required are as follows:

- **Instruction(i).** Callback before the instruction with index i is executed.
- **Transition(i).** Callback when a branch is taken where i is the index of the transition in the control flow graph of the method.
- **Constraint(i, success).** Callback when an assertion or assumption is executed. `success` is a boolean flag that indicates if the constraint holds.
- **Enter(m).** Callback when method m is entered.
- **Leave(m).** Callback when method m is exited.

This execution model is preferred if the language has a runtime environment that is very different from the CLR that Delfy is using. The advantage is that the original source code and all libraries available in the source language can be used.

IR to IL Compilation

If the source language also runs on the CLR, this execution model can save time by directly compiling the program model into CLR types and IL code (the intermediate language used by the CLR). The IL code is then just-in-time compiled and can be executed in the same virtual machine. The advantage of this method is the automatic instrumentation. Since the IL code is generated from the IR representation, the compiler can automatically insert the required callbacks into the compiled code. Furthermore, when the user provides a mechanism to map instructions back to the original source code (e.g. bidirectional source language adapter), the compiler emits debugging information that can be used inside Visual Studio to debug the code while it is being explored.

2.4.6 Out-of-process exploration

A key requirement of the project was the direct integration of Delfy into the development environment of the source language. However, during the exploration of user code, we are constantly compiling and executing code that is still under development and potentially dangerous (e.g. infinite recursion leads to stack overflows). If we would explore the code in the same process as the IDE, we would risk crashes and out-of-memory exceptions from constantly loading additional code. To mitigate this problem, Delfy supports out-of-process exploration (OOPE). When using OOPE, Delfy parses the source code inside the host process and also sends it to an external agent process. The agent process explores the method and sends the exploration report (i.e. the collected results) back. Delfy then merges the exploration report with the already parsed AST, so that the user does not see a difference between local exploration and OOPE. Should the exploration result in a crash, the operation simply times out from the view point of the host process. The agent process can be restarted quickly without losing any information, while a crash of the whole IDE would be fatal.

To allow this, all information that is exchanged between host process and agent must be serialisable. For the exploration request, this is simple because it primarily consists of the source code of the program, which is simply a string. However, the exploration report contains complex objects and numerous references to elements of the program model. These objects all live inside the agent process. When the host process retrieves the exploration report, Delfy automatically replaces all references to objects outside of the exploration report (e.g. references to program model objects) with the corresponding objects inside the host process.

Chapter 3

Integrating Delfy and Dafny

The first part of this report introduced the new Delfy Engine that implements the core algorithms for running dynamic symbolic execution based on a program model that generalises over an arbitrary source language. In this chapter, we focus on integrating the Dafny language with Delfy and examine how some of the more interesting features of Dafny are translated into the Delfy program model.

3.1 Supported Language Features

Feature	original Delfy	Delfy
Classes and Methods	yes	yes
Objects	yes	yes
Quantifiers	yes	yes
Object construction (`new`)	yes	yes
`old` keyword	yes	yes
`fresh` keyword	yes	no
Sets	partially[1][2]	yes[3]
Sequences	partially[1][4]	yes
Uninterpreted methods	partially[1]	yes
Non-deterministic values	partially[1]	yes
Assign-such-that statements	partially[1][5]	yes
Aliasing of arguments	no	yes
Arrays	no	yes
Multidimensional Arrays	no	yes

Table 3.1: Supported Danfy language features

Table 3.1 provides an overview of the Dafny language features that are supported by Delfy and compares the original Delfy with the new Delfy Engine in combination with the Dafny adapter. In general, the new implementation supports a broader spectrum of features or improves existing features to be more generic. The only feature that is not supported anymore is the `fresh` keyword. This keyword allows to detect if an object instance was newly instantiated inside the current method

[1]Only for ints, nats and booleans
[2]Only membership, equality, inequality, union, intersection and difference operations
[3]All operations of the original Delfy plus subset and proper subset relationship, disjointness.
[4]Only length, membership, equality and inequality.
[5]Only if Dafny can statically determine a range of possible values.

or one of its callees. The reason for not supporting this is not a restriction of Delfy itself, but the relatively complex instrumentation that is needed to determine this property in the generated executable code. To migrate this instrumentation code was not possible in the timeframe of the project.

3.2 Translating Dafny to Delfy IR

Dafny is a class-based language making the translation of the type system to the OOP-based Delfy IR relatively simple. Modules and nested modules are mapped to namespaces. Dafny classes are mapped to reference types in Delfy. Dafny supports multiple types of method-like constructs called methods, functions and predicates. All these different concepts are mapped to the generalised method model of Delfy that supports multiple parameters and results.

For most of the Dafny statements and expressions the translation to the Delfy program model is straight forward. However, some of the more interesting language constructs need more attention and a combination of IR instructions to represent. The following sections describe these language features and the implemented solutions.

3.2.1 Specifications

While Dafny supports various types of specifications, Delfy can only handle assumptions and assertions. Preconditions are translated into assumptions at the beginning of a method, while postconditions become assertions at the end of a method.

Loop invariants are asserted after the loop condition is checked and assumed at the beginning of the loop body and after the loop ends. Listing 3.1 illustrates this transformation with pseudo-code. The positioning of the `assert` and `assume` statements is crucial to the correctness of the *input-dependent loop handling* strategy introduced in Section 2.4.4.

```
begin:
  condition = <loop condition>;
  assert <invariant>;
  if !condition goto exit;
  assume <invariant>;
  // Loop body
  goto begin;

exit:
assume <invariant>;
```

Listing 3.1: Translation of while-loop with invariant

3.2.2 Quantifiers

Quantifiers in Dafny allow to assert properties over a bounded range, e.g. given by a lower and upper integer bound, a set, a sequence or booleans. With exception of boolean ranges, which can simply be expressed using a conjunction or disjunction, the quantifier expression is expanded into an inline loop that iterates over all elements inside the range. The implementation is short-circuited, meaning that the iteration is interrupted as a soon as the result is determined.

This implementation was chosen contrary to a solution that extends the Delfy IR with quantifiers because they suffer from many of the same problems that loops do (e.g. input-dependentness). This choice allows strategies that deal with loops also to be effective for quantifiers. Furthermore, it allows for arbitrary code in the expression over which the quantifier is applied. However, it also has the drawback that the code is more general and therefore is more difficult for the constraint solver to deal with.

3.2.3 Sets

Sets were implemented by providing both a special set type for Delfy and a C# implementation of sets for use at runtime that follows the same semantics as the built-in set type of Dafny. In contrast to the previous implementation, immutable data structures were used because they mimic the value-type behaviour of Dafny sets much better and do not require cloning the elements at runtime, which makes the implementation faster.

To formalise sets for constraint solving, the built-in Z3 set-logic was used. There is a potential mismatch between the axiomatisation of sets in Z3 and in Boogie, which could lead to unnecessary test cases or missed bugs in the worst case. However, the axiomatisation is pluggable and could be replaced later with a better axiomatisation, found through experimentation.

Additionally to the membership, equality, inequality, union, intersection and difference operations that are supported by the original Delfy, the new implementation also supports the subset and proper subset relation as well as disjointness. Set cardinality is only supported for code generation, but cannot appear as part of the path condition because Z3 sets have no notion of cardinality.

3.2.4 Arrays

A new feature is the support for arrays and multidimensional arrays. Arrays use the built-in array types of the CLR as runtime representation. To formalise arrays in Z3 a slight adaption of the scheme introduced in Section 2.4.2 is used. The array length is encoded as a field of the array object. For multidimensional arrays a different field for each dimension is used. The array elements are represented using Z3's array theory[6]. The data structure is an array of an array of the element type. The outer array is indexed using the object identity ω and the index for the inner array corresponds to the index of the array element. For multidimensional arrays, the indices are flattened by iteratively adding the index in one dimension and then multiplying with the array length in the next dimension.

This formalisation allows aliasing, nullity of arrays and the appearance of the array length in all dimensions as well as references to any array element as part of the path constraint.

Chapter 4

Combining Verification and Dynamic Symbolic Execution

In this chapter, we present how we combine the information from the static analysis, the exploration data from the dynamic symbolic execution and the results of the static verification to provide different types of feedback to the developer. We also show how this feedback is incorporated into the development environment and visualised for the user.

4.1 Colour-coding of assertions

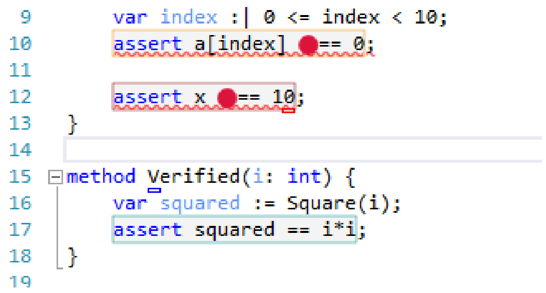
All assertions in the explored part of the code are highlighted in different colours to reflect their status. Figure 4.1 shows how assertions are highlighted using the following colours:

- **Green.** Assertions are highlighted in green if they were either proved by the static verifier or by full exploration of the method.
- **Yellow.** An assertion cannot be proved by the static verifier, but we can also not generate a counterexample that violates the assertion. In this case the developer should try to simplify the code until we can either find a counterexample or the proof can be found.
- **Red.** We found a set of input parameters that leads to an assertion violation. In this case, we provide a *smart tag* that starts a debugging session with a counterexample that results in this assertion violation.

4.2 Proving assertions by exhaustive search

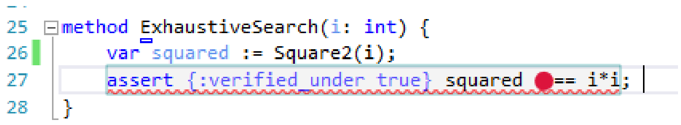
If we can fully explore an assertion by enumerating all possible paths and never run into an assertion violation, we have proven that the method is correct. This is most effective for code that is missing specifications or where the specifications are too weak. Since the prover summarises loops/method calls by their invariant/pre- and post condition, while DSE simply unrolls/inlines the code, DSE can be more effective in these situations.

In Figure 4.2 we see the feedback of the IDE for a code snipped that uses an underspecified function (`Square2`). While Dafny cannot prove the correctness because of a missing post condition, Delfy can show that the assertion always holds.



```
 9        var index :| 0 <= index < 10;
10        assert a[index] == 0;
11
12        assert x == 10;
13    }
14
15    method Verified(i: int) {
16        var squared := Square(i);
17        assert squared == i*i;
18    }
19
```

Figure 4.1: Delfy highlights assertions with different colours to indicate their status (line 10: no counterexample, line 12: counterexample found, line 17: verified). The red dot indicates a verification error.



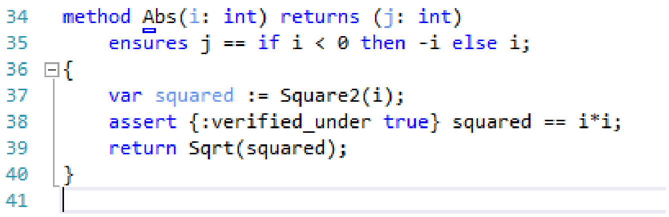
```
method ExhaustiveSearch(i: int) {
    var squared := Square2(i);
    assert {:verified_under true} squared == i*i;
}
```

Figure 4.2: Delfy can prove an assertion, while Dafny displays an error (red dot).

4.3 Rewriting proved assertion to strengthen verification

If DSE can prove an assertion, it inserts an annotation in the source code (`{:verified_under true}`) that tells the static verifier that it does not need to verify this assertion but instead can assume it to be true. In the case where the prover first failed because of a missing specification, it can now continue and prove parts of the code on which it previously failed. Similarly, we can prune exploration paths if we know, that certain assertions are provably correct (there is no need to invert then).

Now we are in the situation where the static verification can benefit from the results of the dynamic symbolic execution and vice versa. Through applying verification and dynamic symbolic execution alternatively, straight-line code with missing or weak specifications can now be verified as seen in Figure 4.3.



```
34  method Abs(i: int) returns (j: int)
35      ensures j == if i < 0 then -i else i;
36  {
37      var squared := Square2(i);
38      assert {:verified_under true} squared == i*i;
39      return Sqrt(squared);
40  }
41
```

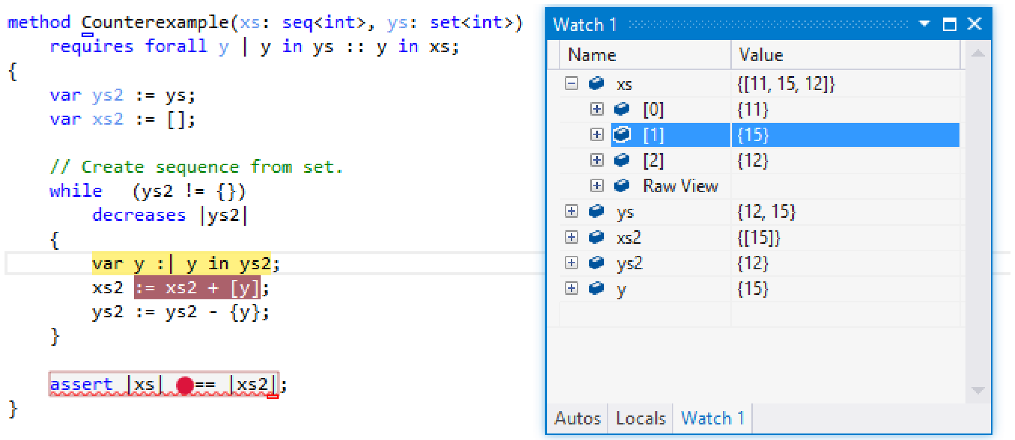
Figure 4.3: Dafny proves the post condition relying of Delfy the prove the assertion in line 38.

4.4 Additional Features

The following features were not core requirements for the wizard and do not use all of the collected information (i.e. a combination of verification and exploration data), but still provide useful and interesting functionality. While mostly existing for convenience, features like the debugging of counterexamples can have a great positive impact on developer productivity.

4.4.1 Debugging of counterexamples

Because Dafny uses the IR to IL compilation and the Danfy adapter works bidirectionally (i.e. it can map elements of the program model back to the source program), we can provide source level debugging of Dafny code including the inspection of variables (watch window) and setting break points as seen in Figure 4.4. By combining this with the ability to persist and rerun previous explorations, it is possible to interactively debug through a failing run step-by-step. This can help developers to find and resolve bugs quickly when a program is too complex to detect a bug simply by looking through the code.



```
method Counterexample(xs: seq<int>, ys: set<int>)
    requires forall y | y in ys :: y in xs;
{
    var ys2 := ys;
    var xs2 := [];

    // Create sequence from set.
    while   (ys2 != {})
        decreases |ys2|
    {
        var y :| y in ys2;
        xs2 := xs2 + [y];
        ys2 := ys2 - {y};
    }

    assert |xs| == |xs2|;
}
```

Watch 1

Name	Value
xs	{[11, 15, 12]}
[0]	{11}
[1]	{15}
[2]	{12}
Raw View	
ys	{12, 15}
xs2	{[15]}
ys2	{12}
y	{15}

Figure 4.4: Live debugging session for Dafny code with breakpoint and watch window.

4.4.2 Visualising the control flow

The control flow of each method is calculated as a result of the static analysis that is performed by the Delfy Engine. Our extension allows to visualise the control flow as an interactive directed graph. The graph shows the individual basic blocks, the possible transitions with their respective condition as well as the invariant state information that is derived by the static analysis. While being especially helpful during the development of the extension itself, there are also applications for educational purposes, e.g. for introduction to program analysis or formal methods.

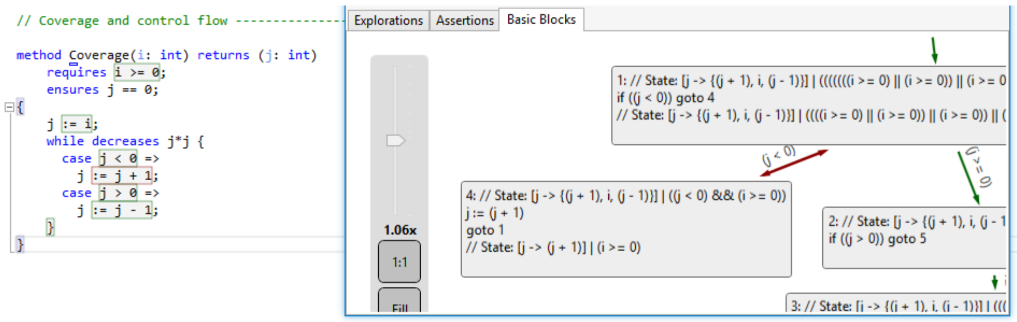
4.4.3 Exploration coverage information

During an exploration, the Delfy Engine uses the callbacks to collect statement and branch coverage information. When the statement coverage flag is enabled, code that was executed during an exploration is highlighted in green, while unreached code is marked in red. This can guide the developer to the parts of the program that are not explorable by Delfy and therefore direct them to areas potentially containing bugs.

The branch coverage is automatically visualised as part of the control flow graph described in the previous section. Before the first exploration is complete, all edges are drawn in black. After an exploration report is available, the edges are coloured green if they were transitioned at least once, otherwise they are coloured red. Figure 4.5 shows how we visualise the control flow as well as statement and branch coverage.

4.4.4 Auto-Exploration

After selecting an entry method for exploration, the source code of the current file is automatically monitored for changes. When the changes in the document stop for a short amount of time and



```
// Coverage and control flow --------------

method Coverage(i: int) returns (j: int)
    requires i >= 0;
    ensures j == 0;
{
    j := i;
    while decreases j*j {
      case j < 0 =>
        j := j + 1;
      case j > 0 =>
        j := j - 1;
    }
}
```

```
1: // State: [j -> {(j + 1), i, (j - 1)}] | (((((((i >= 0) || (i >= 0)) || (i >= 0
if ((j < 0)) goto 4
// State: [j -> {(j + 1), i, (j - 1)}] | ((((i >= 0) || (i >= 0)) || (i >= 0)) || (

4: // State: [j -> {(j + 1), i, (j - 1)}] | ((j < 0) && (i >= 0))
j := (j + 1)
goto 1
// State: [j -> (j + 1)] | (i >= 0)

2: // State: [j -> {(j + 1), i, (j - 1
if ((j > 0)) goto 5

3: // State: [j -> {(j + 1), i, (j - 1)}] | (((
```

Figure 4.5: Visualisation of control flow and coverage information.

the source code does not contain any syntax errors, the method is automatically reexplored in the background and all the annotations are updated. Even if the document changes during the exploration, the annotations are displayed at the correct position based on the change history since the start of the last exploration. If auto-exploration is not desired, e.g. because a method is still in early development, it can be disabled manually.

Chapter 5

Related Work

Various other approaches to combining verification with systematic testing already exist. YOGI[16] is similar to our work in that it searches for proof of a specific property and also tries to find a test that violates this property. However, the coupling between verification and test generation is bidirectional and thus tighter than in our approach, where the feedback only flows from Delfy to Dafny. The idea of combining unsound static checkers with dynamic test generation is explored in works like Check'n'Crash [3], DSD-Crasher [4], DyTa [9].

Dafny[13] is a programming language with built-in specification constructs to support sound static verification. The Dafny verifier is preferably run in an integrated development environment (IDE)[14, 15], which extends Microsoft Visual Studio. Similar to our wizard, it displays verification errors inside the source code editor.

Boogie[1] is an Intermediate Verification Language (IVL) for describing proof obligations to be discharged by a reasoning engine, typically an SMT solver. Boogie is similar to the Delfy Engine by targeting various source languages using an intermediate abstraction. While Delfy focuses on dynamic symbolic execution, Boogie is aimed at static verification.

Pex[18] is a white-box testing tool based on dynamic symbolic execution that automatically explores methods and generates test-cases. Pex works on CIL level and can therefore also be used on the compiled Dafny code. A comparison between Pex and the original Delfy can be found in Patrick Spettel's master thesis[17].

IronClad[11], developed by Microsoft Research, lets a user securely transmit their data to a remote machine with the guarantee that every instruction executed on that machine adheres to a formal abstract specification of the app's behaviour. One of the key verification tools used in its development is Dafny.

Chapter 6

Conclusion

In this project we developed a language-agnostic engine for dynamic symbolic execution based on a model of the source program. The chosen architecture for this engine provides a high flexibility for modelling different type systems and has various extensibility points, not only to support complex source language constructs, but also to substitute and improve various aspects of the exploration algorithm itself. Out of the box, Delfy supports subtyping, inheritance, value types, reference types with heap state and aliasing. Performance and stability were considered from the start and manifest themselves in features like concurrent and out-of-process exploration.

We use this engine to provide dynamic symbolic execution for Dafny and are able to support more language features and achieve shorter exploration times than an already existing implementation. By combining the information from the verifier with the results of the program exploration, we can help the user during development by providing counterexamples, proving properties of straight-line code and prioritising assertions based on their path coverage. All this information is visualised directly inside the development environment and by exploring the code automatically in the background, the developer receives fast and reliable feedback. Additional features like debugging based on counterexamples and coverage information further enhance the development process.

We see the potential applications of the wizard mostly in the area of eduction and research, e.g. for the IronClad project that already uses Dafny as its main verification tool.

References

[1] Michael 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*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[2] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335. ACM, 2006.

[3] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE*, pages 422–431. ACM, 2005.

[4] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *TOSEM*, 17:1–37, 2008.

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

[6] Leonardo de Moura and Nikolaj Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52. IEEE Computer Society, 2009.

[7] Patrick Emmisberger. Dynamic test generation with static fields and initializers. Bachelor's thesis, ETH Zürich, Switzerland, July 2013.

[8] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *SAC*, pages 2103–2110. ACM, 2010.

[9] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. DyTa: Dynamic symbolic execution guided with static verification results. In *ICSE*, pages 992–994. ACM, 2011.

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

[11] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, October 2014. USENIX Association.

[12] Claire Le Goues, K. Rustan M. Leino, and MichałMoskal. The Boogie verification debugger. In *SEFM*, volume 7041 of *LNCS*, pages 407–414. Springer, 2011.

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

[14] K. Rustan M. Leino and Valentin Wüstholz. The Dafny integrated development environment. In *Formal-IDE*, volume 149 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–15. Open Publishing Association, 2014.

[15] K. Rustan M. Leino and Valentin Wüstholz. Fine-grained caching of verification results. In *CAV*, LNCS. Springer, 2015. To appear.

[16] Aditya V. Nori, Sriram K. Rajamani, Saideep Tetali, and Aditya V. Thakur. The YOGI project: Software property checking via static analysis and testing. In *TACAS*, volume 5505 of *LNCS*, pages 178–181. Springer, 2009.

[17] Patrick Spettel. Delfy: Dynamic test generation for dafny. Master's thesis, ETH Zürich, Switzerland, September 2013.

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

[19] Dries Vanoverberghe, Nikolaj Bjørner, Jonathan Halleux, Wolfram Schulte, and Nikolai Tillmann. Using dynamic symbolic execution to improve deductive verification. In *Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 9–25. Springer, 2008.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Integrating dynamic test generation with sound verification

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Emmisberger	Patrick

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Grüningen, 2. Juli 2015	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.