Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Fuzzing the Dafny Verifier

Author:
Wei Yi Tee

Supervisor:
Prof. Alastair Donaldson

Second Marker:
Prof. Cristian Cadar

June 19, 2023

Abstract

Dafny is a programming language equipped with a static program verifier. The verifier checks the correctness of Dafny programs relative to well-known safety criteria e.g. bounds safety and explicit specifications provided by the user. Development in Dafny is attractive as code will be provably correct. The correctness guarantees of Dafny programs relies on the correctness of the verifier. Hence, it is important to ensure that the Dafny verifier is thoroughly tested and bug-free.

In this project, we present our experience in developing a mutation-based fuzzer targeted at the Dafny verifier. Taking inspiration from existing mutation techniques, we design and implement a range of equivalence mutation techniques that will generate new variants from an existing Dafny program. These variants are used to stress-test the Dafny verifier in hopes of uncovering a bug. Finally, we present an evaluation of the effectiveness of our fuzzer.

Acknowledgements

To my supervisor, Professor Alastair Donaldson, for the opportunity to work on this project and his guidance. To my second marker, Professor Cristian Cadar, for the Software Reliability module which was challenging but interesting.

To my friends, for their companionship through the best and worst of university. To the unnamed people I've encountered during these 4 years that have inspired me to improve.

To my family, no words are enough to express my thanks. Thank you for always being there.

To myself, for making it through.

Contents

Chapter 1

Introduction

1.1 Motivation and Problem Statement

The integration of program analysis and verification into the development workflow is on the rise for many kinds of software. These tools help programmers detect bugs before their code goes to production, and helps increase confidence in the software.

Dafny [1] provides an environment to develop verified code, thanks to its in-built verifier. The presence of multiple back-ends that can compile a Dafny program to different target languages e.g. C#, Java and Python makes it easy to integrate the use of Dafny into existing workflows. These attractive features lead to the increased adoption of Dafny.

With all program analysis and verification tools, the correctness of the tool itself is critical for the guarantees the tool claims it provides to hold. A bug in the tool can lead to bugs in the software developed using the tool, hence it is crucial to ensure that the tool is well-tested for bugs.

For such complex pieces of software, manual testing is usually insufficient to test all behaviour of the tool. Fuzz testing is a popular bug-detection approach where tests (in our context, programs) are automatically created to stress the system under test with. We look at how we can adapt fuzzing for the Dafny verifier.

The automatic construction of programs lies at the core of fuzzing for which there are two main classes of approach. Generation-based approaches construct a program from scratch, whilst its less popular counterpart - mutation-based approaches construct variants from an existing program. We refer to the original program as the seed, and its variants as mutants. With the construction of Dafny programs, a challenge lies in the creation of program specifications which is not trivial to automate as it often requires a high-level understanding of the program's intentions. This motivates us to pivot towards mutation-based approaches for constructing Dafny programs as it will allow us to make use of existing specifications.

The test oracle problem describes the challenge of determining the ground truth of a test case in fuzzing. The common workaround for this issue includes differential testing which cross-checks the results returned from different versions of the software, and metamorphic testing which cross-checks the results on a related group of tests. Equivalence testing is a subset of metamorphic testing that cross-checks the results of a set of equivalent programs. Since the programs are equivalent, they should return the same result. Because there is only a single Dafny verifier, we pivot towards the equivalence testing approach.

What remains of this equation are the mutation techniques for constructing equivalent variants from a program. We explore this in our project.

1.2 Contributions

1. We create a mutation-based fuzzer for equivalence testing of the Dafny verifier.
2. We design and implement a diverse and unique set of equivalence mutations. Additionally, we propose extensions upon the existing set of mutations.
3. We evaluate the capabilities of our fuzzer compared to existing fuzzers.

1.3 Ethical Considerations

Our fuzzer aims to find bugs so that they can be fixed or made aware of to users. However, malicious users may use the tool to find and exploit vulnerabilities in software. This is especially important in the context of Dafny which is often used to write critical software. Exploiting bugs in such software may cause large losses.

1.4 Report Structure

Chapter 2 describes the Dafny features that we will encounter in this paper to provide context.

Chapter 3 introduces fuzzing which is a core concept for our project.

Chapter 4 analyses other existing work relevant to fuzzing Dafny or mutation-based fuzzing.

Chapter 5 studies Dafny and presents how we concretise the scope of our project.

Chapter 6, 7 discusses the design and implementation for our mutation-based fuzzer and selected mutations.

Chapter 8 discusses the evaluation of our mutation-based fuzzer.

Chapter 9 concludes our project and provides ideas for future work.

Chapter 2

Background - Dafny

Dafny is a programming language that provides support for formal specification in addition to usual imperative and functional features. It comes with an in-built verifier that checks the correctness of the Dafny program with respect to general safety criteria and user provided specifications.

This chapter describes features and concepts in Dafny's programming language that are relevant for the project. Further information on Dafny can be found in the reference manual [2].

2.1 Types

A type in Dafny can be classified as an immutable value type or a mutable reference type.

Immutable value types are independent of the heap. Examples of immutable value types that we will come across in this paper include:

- Built-in scalar types: `bool`, `char`, `int`, `real`
- Built-in collection types: `set`, `multiset`, `seq`, `map`, `string` (`seq<char>`)
- Inductive datatypes
- Subset types based on value types

Mutable reference types are stored in the heap. Examples of mutable reference types include the `class` and `array` type.

2.1.1 Map types

A `map<T, U>` is an immutable collection where values of type `T` map to values of type `U`.

To construct a map, we may use a map display expression. A map display expression is formed by the `map` keyword followed by a possibly empty comma-separated list of entries of the form `t := u` enclosed in square brackets. `t` and `u` are expressions of the key type and value type. Examples are presented in Figure 2.1.

```
1 map[ ]  // empty
2 map['a' := 1, 'b' := 2]
```

Figure 2.1: Map display expression

Table 2.1 presents the map operations that we will encounter in this paper.

`t in m`	Checks that the key `t` is present in `m`
`m[t]`	Returns the value indexed by `t` in `m`
`m[t := u]`	**Map update expression**. Creates a new map containing all entries of `m` except for the entry indexed by key `t` whose value will be `u`. Only a single update is allowed.
`m + n`	**Map merge expression**. Creates a map containing entries from both maps `m` and `n`. If the same key exists in both maps, its value in the returned map will take the value in the latter map which is `n` in this case.

Table 2.1: Map operations

2.1.2 Datatypes

An inductive datatype is declared with the keyword `datatype` with a non-empty '`|`' separated list of datatype constructors. In Figure 2.2, line 1 presents the declaration of the datatype `OptionalInt` with a `None` constructor and a `Some` constructor that takes an integer argument. Line 2 shows how a datatype value of `OptionalInt` can be constructed.

```
1 datatype OptionalInt = None | Some(v: int)
2 var x := None
3 var y := Some(1)
```

Figure 2.2: Datatype

Table 2.2 presents the datatype operations that we will encounter in this paper.

`x.None?`	**Discriminator**. A discriminator is automatically created for each constructor of the datatype. It checks if the datatype value `x` was constructed via the corresponding constructor, in this case `None`.
`y.v`	**Destructor**. Accesses the parameter `v` in `y`.
`y.(v := i)`	**Datatype update expression**. Creates a new datatype value identical to `y` except for the parameter `v` whose value will be `i`. Multiple updates are allowed.

Table 2.2: Datatype operations

2.1.3 Subset types

"A subset type is a restricted use of an existing type, called the base type of the subset type."

Figure 2.3 declares a subset type `Positive` whose values are of base type of `int` constrained to be greater than `0`. The witness provides an example value for the type which is `1` in this example.

```
1 type Positive = i: int | i > 0 witness 1
```

Figure 2.3: Subset type

2.1.4 Auto-initialisable types

Dafny provides the concept of auto-initialisable types. For the purposes of this project, it is sufficient to interpret an auto-initialisable type as a type that the Dafny compiler knows how to

generate a default value for. An example of an auto-initialisable type is the `int` type whose default value is `0`.

Variables of auto-initialisable types can be used before an explicit initialisation or assignment. Using variables of non-auto-initialisable types without an explicit initialisation or assignment will generate a warning.

2.2 Specifications

Specifications provide Dafny with hints about important properties of the program state to aid verification. A specification consists of zero or more clauses. There are different types of specification clauses: requires, ensures, decreases, modifies, reads, invariant.

2.2.1 Requires

Precondition for functions and methods. For all calls, Dafny checks that the precondition is satisfied. The callee can assume that the preconditions upon entering the function or method.

2.2.2 Ensures

Post condition for functions and methods. Dafny checks the post condition is satisfied on all exits from the function or method. The caller can assume that the postconditions hold when the call returns.

2.2.3 Modifies

Write frame for methods and loops. The set of heap locations that the construct is allowed to write to.

2.2.4 Reads

Read frame for functions. The set of heap locations that the function is allowed to read.

2.2.5 Decreases

Termination metrics for loops, functions and methods. Dafny checks that the metric decreases on each loop iteration / recursion to guarantee termination.

2.2.6 Invariant

A property that must hold upon every loop test (check of the loop guard). Dafny checks that the invariant holds right before and right after each loop iteration.

2.3 Functions

"A pure mathematical function, it is allowed to read memory specified in its reads clause but is not allowed to have any side effects." Figure 2.4 presents an example of a function declaration.

The function will return a single result.

The function body is a side-effect free expression.

```
function f(a: array<int>, i: int): int
  requires 0 <= i < a.Length
  reads a
{ a[i] }
```

Figure 2.4: Function declaration

The function specification consists of 0 or more requires, ensures, reads or decreases clauses.

2.4 Methods

Figure 2.5 presents an example of a method declaration.

```
method m(a: array<int>, i: int)
  requires 0 <= i < a.Length
  ensures a[i] == 0
  modifies a
{ a[i] := 0; }
```

Figure 2.5: Method declaration

The method may have 0 or more return values.

The method body is a list of potentially side-effecting statements.

The method specification consists of 0 or more requires, ensures, modifies, or decreases clauses. Methods are allowed to read anything in the heap, thus reads clauses are not needed.

2.5 Statements

2.5.1 Assignment Statements

Figure 2.6 is an example of an assignment statement.

```
x, y := 1, 2;
```

Figure 2.6: Assignment statement

Dafny supports parallel assignment where multiple variables may be updated in the same statement. The right-hand-side values must not be method calls which are to be handled in call statements.

2.5.2 Call Statements

Lines 3 - 4 in Figure 2.7 are examples of call statements.

Although the syntax on line 3 is similar to assignment statements, the difference lies in the fact that the right-hand-side of a call statement is a method call. Assignment statements do not allow method calls on the right-hand-side. There may be 0 or more expressions on the left-hand-side of a call statement which must match the number of return values of the function. For each call statement, we can only invoke a single method call.

```
1 method m() returns (x: int, y: int) { return 1, 2; }
2 method n() { }
3 x, y := m();
4 n();
```

Figure 2.7: Call statement

2.5.3 Declaration Statements

Lines 2 - 4 in Figure 2.8 are examples of declaration statements. Declaration statements may have initialisers which correspond to assignment or call statements.

```
1 method m() returns (x: int, y: int) { return 1, 2; }
2 var v: int;
3 var w, x := 1, 2;
4 var y, z := m();
```

Figure 2.8: Declaration statement

2.5.4 While Loop Statements

Figure 2.9 is an example of a while loop statement.

```
1 while i < n {
2   i := i + 1;
3 }
```

Figure 2.9: While loop statement

2.5.5 For Loop Statements

Figure 2.10 is an example of a for loop statement.

`i` is the loop index. The initial `0` and final `n` values of the loop index are referred to as loop-start and loop-end. Dafny also supports decreasing for loops by replacing the `to` keyword with `downto` e.g. `for i := n downto 0 { }`.

```
1 for i := 0 to n {
2   print i;
3 }
```

Figure 2.10: For loop statement

2.5.6 Break and Continue Statements

Break and continue statements may be labelled or non-labelled.

Labelled break statements as in Figure 2.11a, line 3 are required to be enclosed in the statement with the label. We refer to this statement as the target statement. The break statement will exit the target statement, and jump to the next statement after the target statement. Continue statements may only be labelled if the label refers to a loop. In this case, the continue statement will continue the target loop.

Non labelled break statements as in Figure 2.11b, line 4 may have multiple break occurrences and are required to be enclosed in loops. The break statement will break as many loops as the number of `break` occurrences in the statement. In the example, the `break break` will exit both loops. A `break continue` will break the inner loop and continue the outer loop.

```
1 label x:
2 if true {
3   break x;
4 }
```

(a) Labelled

```
1 for i := 0 to m {
2   for j := 0 to n {
3     if i + j == 10 {
4       break break;
5     }
6   }
7 }
```

(b) Unlabelled

Figure 2.11: Break and continue statement

Chapter 3

Background - Fuzzing

This chapter defines fuzzing and discusses different concepts within the domain of fuzzing.

3.1 Motivation

There are different approaches to ensure the correctness of a software.

Manual testing involves writing tests to check that the software behaves as expected. A limitation is that it may not be able to capture all behaviour a software may exhibit. This is especially true for complex pieces of software with infinite ways of behaving. Additionally, manual testing incurs a maintenance cost.

Formal verification checks that a software is correct via formal methods. An example of a formally verified compiler is CompCert [3]. In [4], it is shown that CompCert is more resistant to fuzzing compared to other non formally-verified compilers. Despite the strong guarantees presented with this approach, formal verification is less adopted due to the technical effort and complexity in implementing formally verified software.

Fuzzing is an approach that avoids the limitations of both approaches. When implemented effectively, fuzzing ensures that a large subset of the software's behaviour is tested at a reasonable technical cost.

3.2 Definition

Fuzzing is an automated testing approach where the system under test is invoked on a large number of random tests and its behaviour under these tests are observed. The idea is that by exposing the system to a diversity of inputs, we hope to find some test that will trigger and allow us to detect misbehaviours in the system that can then be fixed.

Fuzzing is attractive as it automates test-case construction. This is true particularly for complex pieces of software such as compilers, SAT/SMT solvers, program analysers. These classes of software exhibit an infinite amount of behaviour and it is unsustainable to write tests manually to cover enough of the codebase.

Within a fuzzer, there are two core issues to address - (1) the automatic construction of tests, (2) determining the expected outcome for a specific test.

For the context of our project, the test will take the form of a program. Hence, we base the rest of our discussion on programs.

3.3 Test Construction

The two main classes of program construction approaches are generation-based approaches and mutation-based approaches.

3.3.1 Generation-based Approach

A generation-based approach creates programs from scratch, usually with knowledge of the grammar of the input language. This approach usually involves the repeated random selection of production rules to construct the AST. Upon constructing an AST node, we can choose to fill it in with existing elements, or create new elements that are filled out later. Well-known program generators include Csmith [5] and YARPGen [6].

3.3.2 Mutation-based Approach

A mutation-based approach applies changes to existing programs to derive new programs. An example of a mutation-based compiler fuzzer is Orion [4] which generates equivalent mutants from a program by randomly pruning sections of dead code. We investigate various mutation-based approaches at greater detail in sections 4.2, 4.3.

3.4 Test Oracle

For the automatically constructed tests, we need a means of determining the expected outcome of running the test with the system under test. This is known as the test oracle problem. We discuss the common approaches to address this problem.

3.4.1 Differential Testing

Differential testing [7, 8] involves cross-checking the outcomes returned from running different versions of a software on the same test. The outcomes should match, otherwise there is a bug in at least one implementation. Furthermore, we define bugs as deviant behaviour [9] - this states that the majority outcome is the correct outcome. Any different outcome indicates a bug in the implementation.

In compiler testing, there are three strategies that may be used: cross-compiler (across different compilers for a language), cross-optimisation (across different optimisation levels of a compiler) and cross-version (across different versions of a compiler). The cross-optimisation strategy is especially interesting as it allows investigation into the complex interactions between different optimisations.

A limitation of differential testing is that it requires the existence of multiple implementations for the type of software being tested.

3.4.2 Metamorphic Testing

Metamorphic testing [10] involves the use of metamorphic relations. These relations describe the expected change in the output given some change in the input. We can test the system by checking if the expected change holds for a particular change in the input. If the expected change is not observed, we have a potential bug in the implementation. In this approach, we don't have to know the exact values of the outcome.

We demonstrate an example taken from [11] for the `sin` function. Since we know that `sin(2pi + x) = sin(x)`, we can test implementations for a `sin` function by giving these two inputs to the program and verifying that they are the same. We don't need to know what `sin(x)` is expected to evaluate to.

Equivalence Modulo Inputs [4] is a form of metamorphic testing using equivalence relations. A set of programs is said to be equivalent modulo the input set `I` if they show equivalent behaviour under all values in `I`. EMI testing involves cross-checking EMI programs and ensuring that the outcomes returned match.The `sin` example presented is an example of EMI.

3.4.3 Use of Profiling

The execution of a program is profiled to extract information about values of variables. In the context of analysis and verification tools, the information collected from profiling may be used to construct assertions with a known truth value. We compare the verification outcome of the assertions returned from the tool with the expected outcome we inferred from profiling.

For example, in MCFuzz [12], the input program is instrumented with counter variables corresponding to each branch of the code, the counters will be updated at their respective branches. After executing the program, the final values of these counters are retrieved to form assertions about how often a branch is reached. This approach is also used in XDsmith [13] which we describe in Section 4.1.

This approach requires that the programs checked are deterministic so that the program state and the derived expected outcome of assertions does not change over runs. Additionally, this approach relies on the compilation and execution of the program being correct.

3.5 Bug Classification

In the context of compilers, a bug may be classified as a crash or a miscompilation. A crash terminates the compilation process. A miscompilation bug does not terminate the compiler, rather it generates code that is wrong silently.

In the context of solvers, analysis and verification tools, a bug may be classified as a soundness bug or a precision bug. Soundness bugs, also known as false negatives, occur when the tool fails to warn on an incorrect example. Precision bugs, also known as false positives, occur when the tool warns incorrectly about a correct example.

Chapter 4

Related Work

This chapter explores related works on fuzzing. We hope to derive useful insights from existing experiences to guide the design of our mutations.

Section 4.1 describes XDSmith - a generation-based fuzzer for Dafny.

The lack of work on mutation-based fuzzing for verification tools leads us to take inspiration from fuzzers for neighbouring classes of software, namely SMT solvers and compilers. This is explored in Section 4.2, 4.3.

4.1 XDsmith

Presented in [13], XDsmith is a program generator for a subset of Dafny.

Generation of programs in Dafny happens in two stages. The first stage generates unannotated programs using an off-the-shelf tool. The program is injected with print statements and executed to collect information about the program state at various program points. XDsmith restricts the program to not take any inputs which ensures that the execution is deterministic.

The second stage uses the knowledge collected from the program execution to generate specifications that are injected into the program. Figure 4.1 demonstrates a simplified example of how an annotated program is derived from profiling the unannotated program's execution.

XDsmith generated programs are loop-free, recursion-free, heap-free, and quantifier-free. This is to reduce the complexity of creating specifications and determining verification outcomes. The restriction on loops and recursion is to guarantee termination. The use of the heap and quantifiers are excluded to simplify reasoning on the program.

4.2 Mutation-based Fuzzing for SMT Solvers

Presented in [14], OpFuzz is a fuzzer for SMT solvers based on applying type-aware operator mutations to seed formulas. This involves replacing an operator within a seed formula with another operator of the same type to derive another well-typed formula. The type of an operator in the SMT-LIB language is derived from its input and output types. We show an example in Figure 4.2 that demonstrates a mutation between comparison operators.

Since a seed formula has a finite number of operators, and each operator has a finite number of candidates for replacement, the mutation space that OpFuzz allows for is restricted. In [15], TypeFuzz introduces the generative type-aware operator mutation technique to address OpFuzz's finite mutation space problem. The idea is to replace an expression within the seed formula with a new expression generated from existing symbols. The expression generation process involves first selecting an operator of a type conforming to the expression that will be replaced, and then selecting expressions from the formula as its operands. Figure 4.3 presents an example of a seed

```
method ReturnSelf(x: int)
  returns (y: int)
{
    y := x;
    print y; // 1
}

method Main() {
    var x := ReturnSelf(1);
    print x + 5; // 6
}
```

(a) Unannotated

```
method ReturnSelf(x: int)
  returns (y: int)
  requires x == 1 // restrict input
  ensures y == 1
{
    y := x;
    print y;
    assert y - 1 == 0; // true, x is 1
}

method Main() {
    var x := ReturnSelf(1);
    print x < 32;
    assert (x + 5) * 5 > 25; // true, x is 6
}
```

(b) Annotated

Figure 4.1: XDsmith - Generating annotated programs by profiling unannotated programs

```
(>= x x)
```

(a) Original

```
(< x x)
```

(b) Mutant

Figure 4.2: OpFuzz - Type-aware operator mutation for SMT formulas

```
(str.replace x "B" (str.++ "B" "B"))
```

(a) Original

```
(str.replace x (str.++ x x) (str.++ "B" "B"))
```

(b) Mutant

Figure 4.3: TypeFuzz - Generative type-aware operator mutation for SMT formulas

formula and its mutant derived via this approach. The second operand of type `string` in `str.replace` is replaced by a freshly generated string concatenation operation of two string expressions from the formula.

4.3 Mutation-based Fuzzing for Compilers

All fuzzers explored in this section adopt equivalence testing. Hence, all mutations presented here are equivalence mutations. Table 4.1 presents a summary of the fuzzers, its target, and the mutations implemented in the fuzzer.

Fuzzer	**Target**	**Mutations**
Orion [4]	C compilers	dead
Athena [16]	C compilers	dead
Hermes [17]	C compilers	dead, live-inj, wrap
GraphicsFuzz [11]	Graphics shader compilers	dead, live-inj, wrap, vector, identity
SLEMI [18]	Simulink compiler	dead, extract

Table 4.1: Investigated mutation-based fuzzers for compilers.

.

4.3.1 Mutations

[dead] Dead region mutations.

Dead regions refer to parts of a program that are never executed. Hence, modifications to such regions should not change the outcomes we observe from the program.

Orion removes unexecuted statements randomly, which is a simple strategy but has been shown to be able to detect bugs. However, the amount of code available for deletion in a program is finite, hence Orion can only generate a limited number of unique mutants. Athena addresses Orion's issue of having a restricted mutation space by supporting code insertion into dead regions.

In Orion, Athena and SLEMI, the seed program was profiled to determine existing dead regions to modify. Alternatively, new dead regions can be created by wrapping code snippets in a guard that always evaluates to false, as done in GraphicsFuzz and Hermes. This can be useful when profiling is unavailable or expensive.

Although effective, dead region mutations still face some core limitations. Firstly, optimising compilers may remove dead code, such removed code will not be able to test the compiler. Additionally, dead region mutations are only capable of catching compile-time crashes. Miscompilation bugs in the dead region cannot be detected as the code is never executed.

Given such limitations, it is important that mutations are extended to live code. In the remainder of this section, we will look at different examples of live code mutations.

[live-inj] Live code injection.

With live code injection, the idea is to insert code that can be executed but will not change the overall semantics of the program.

In GraphicsFuzz, arbitrary code snippets are extracted from existing programs, variables are then renamed to ensure that the state modified by the code snippet is disjoint to the original program state.

Hermes reverses any side-effects of executing the injected code by saving the program state before entering the injected code region, and restoring the state upon exiting the region. The injected code will follow the template illustrated in Figure 4.4. The value of the variable that is modified by the injected snippet is saved on line 3. Lines 4 - 7 modify and use the variable. Line 9 restores the value of the variable.

```
1  int backup_v = ⟨synthesized valid expression⟩;
2  if (⟨synthesized true predicate⟩) {
3    backup_v = v;
4    v = ⟨synthesized valid expression⟩;
5    if/while(⟨synthesized false predicate⟩) {
6      print v;
7    }
8  }
9  v = backup_v;
```

Figure 4.4: Template for live code injection in Hermes

[wrap] Always-true guard.

Wrapping code in a guard that always evaluates to true.

[vector] Vectorisation.

Variables are merged into a vector. References to the variable are replaced by vector accesses.

[identity] Identity functions.

Mathematical and logical identities are used to transform expressions into equivalent forms.

[extract] Extraction.

Implemented in SLEMI, this is equivalent to extracting blocks of code into a function for procedural languages.

4.3.2 Technicalities

Sourcing code snippets for injection.

Code injected into the program are either sourced from existing programs or constructed via generation techniques.

When programs are sourced from existing programs, such as in Athena and GraphicsFuzz, it is important to ensure that the program is still valid. For user-defined variables, types and functions used in the snippet, it is important to add the relevant declarations. Additionally, duplicate names will need to be changed and breaks out of loops need to be removed.

Use of opaque values.

Opaque values refer to values that are not known to the compiler. This concept is crucial in these mutations targeted at optimising compilers.

If a compiler can infer the value of an element in the program, it can carry out optimisations such as constant propagation and code elimination. This is bad as it may erase away our mutations that are then effectively meaningless. For example, when implementing a mutation that wraps dead code in a guard that always evaluates to false, it is important the compiler doesn't know that the guard is false, otherwise the injected dead code block will be removed.

Generating these values that need to be known by us but not the compiler often involves using pieces of known state in the program and composing expressions. For example, if we know that

`x` is `1` by specifying it as an argument to the program, we can construct the `true` expression as `x == 1`. The compiler cannot detect this because `x` is dependent on the user argument. Runs on the program will however need to be restricted to the specific user argument.

4.3.3 Analysis

We observe that the mutations are targeted at compiler optimisations.

Techniques such as code injection and removal, control flow wrapping alter the control flow graph of a program. Since a majority of compiler optimisations are computed based on control flow graphs, these techniques would prove effective at fuzzing these optimisations.

Techniques such as vectorisation, identity functions and extractions may affect inference of variable values as some level of indirection is introduced. Additionally, they may exercise parts of the compiler which tries to rewrite expensive operations as cheaper operations e.g. function inlining, replacing divisions.

Chapter 5

Designing for Dafny

In this chapter, we develop a better understanding of Dafny. We discuss how it helps us select and build upon the variety of fuzzing techniques explored in the previous chapters for an effective shot at our objective of testing the Dafny verifier.

5.1 Understanding Dafny

We present a high level understanding of Dafny upon inspecting parts of the Dafny codebase.

For verification, there are two stages in the Dafny toolchain that are of interest - resolution and verification. Resolution carries out a significant amount of preprocessing on the Dafny AST which includes name resolution, type inference, specification inference. Verification of a Dafny program involves translating the resolved AST into a Boogie program. The Boogie program is verified using the underlying Z3 SMT solver.

We note that verification, more specifically the translation from Dafny to Boogie, is rather straightforward. There is a high overlap between the features supported in Dafny and in Boogie, a notable amount of the translation is one-to-one. In contrast, the processes occurring during resolution are rather complex.

5.2 Historic Issues

We study some of Dafny's historic issues which are public in Dafny's repository to get an insight to the type of bugs that occur in Dafny and bug-prone locations within the toolchain.

We observe that many bugs are attributed to errors that occur in resolution, e.g. during type inference but don't show up until later in the verifier or compiler.

Additionally, we observe that for a certain class of bugs, they have a pattern of showing up in one feature but not others. For example, Figure 5.1a demonstrates a historic issue [19] where the type constraint of a datatype update expression is not checked due to a type inference bug. On line 12, it is illegal to set the field `v` to a potentially non-zero value due to the constraint on line 1. However, this error bypasses the verifier. The indirection provided via the function call `launder` and the map `rep` are required to trigger the bug.

Figure 5.1b presents a closely related issue [20]. It is another type inference bug but for map update expressions. The type constraint is not checked during the update on line 6. Note that despite the similarities in the issues, the latter issue was only discovered some time after the first issue was corrected.

```
type R = r:R_ | r.v == 0 witness R(0)

datatype S = S(
    rep:map<nat,R>
) {
    function launder(r:R):R { r }

    function cpy(r:nat,v:nat):R
      requires r in rep
    {
        // Does not comply with rule of R if v != 0
        launder(rep[r].(v := v))
    }
    static method bad() ensures false {
        ghost var s := S(map[0 := R(0)]).cpy(0,1);
        assert s.v == 1;
    }
}
```

(a) Datatype update expression

```
type foo = m: map<int, int> | forall n <- m.Keys :: m[n] < 5

function addToFoo(m: foo): foo
  ensures false
{
  m[1 := 7] // Violates rule for foo type
}
```

(b) Map update expression

Figure 5.1: Type inference bug for update expressions

5.3 Discussion

We present the thought process in designing our fuzzer for the Dafny verifier that involves merging together general fuzzing concepts and domain-specific knowledge of Dafny.

Mutation-based fuzzer for equivalence testing

Automating the creation of meaningful specifications in Dafny programs is non-trivial. Therefore, we choose to adopt a mutation-based approach. This will allow us to make use of existing specifications in seeds instead of having to construct them from scratch.

For the test oracle issue, we choose to adopt equivalence testing. Differential testing is out of scope as there is only a single Dafny verifier. Using information collected via profiling to derive the expected verification outcome is only applicable if the program is restricted. Verifiers attempt to check all execution traces, whilst a single profile run checks only one execution trace. Hence, for a single profile, the derived verification outcome will only apply for the program restricted to the corresponding trace. Trying to derive the overall verification outcome by profiling all possible traces is infeasible with infinite input domains and costly to implement.

Combining these two decisions narrows down the scope of our project to an investigation of equivalence mutations for Dafny programs. By equivalence mutations, we mean mutations that will generate variants that are equivalent to the original program.

Designing equivalence mutations

We hypothesise that the existing mutations described in Section 4.3 that are targeted at compiler optimisations will not be effective due to the difference in the purpose and nature of the fuzz target. Compiler optimisations are mostly CFG-based and complex. With the stages of the Dafny verifier that we are interested in - verification is AST-based and straightforward. Resolution is also AST-based and we believe it is less complex than the implementation of compiler optimisations.

The findings presented in Section 5.2 is a major source of inspiration for our mutations. The phenomenon that different outcomes are observed under the same logical scenario indicates that there must be a different implementation for the Dafny features that produced the different outcomes. This gives us the idea to explore mutations that translate between different Dafny features. This mutation would verify that the implementations across different Dafny features have a consistent behaviour.

The final set of mutations that we choose to explore are presented in Chapter 7.

Chapter 6

Implementation - Fuzzer

We build a mutation-based fuzzer based on equivalence mutations. The fuzzer is implemented by wrapping a fuzzing workflow around a mutant generator. The fuzzing workflow integrates different services together and cross-checks outcomes returned from verification for bug-detection. The mutant generator produces equivalent mutants from seeds.

6.1 Fuzzing Workflow

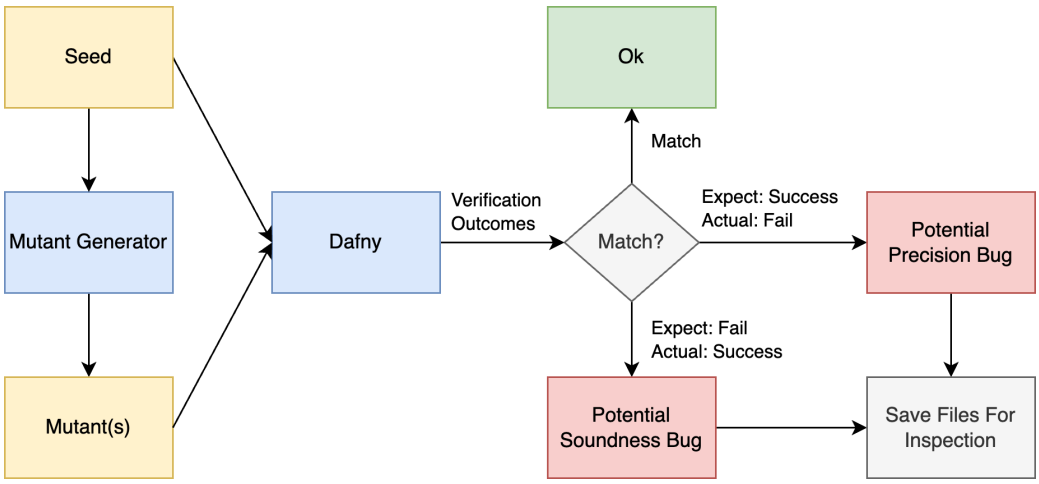

Figure 6.1: Fuzzing workflow

The fuzzing workflow is implemented as a Python script for its simplicity. We describe the steps involved in the workflow, aided with the illustration in Figure 6.1.

First, a seed is randomly picked from an existing corpus. We invoke the Dafny verifier on the seed and record its verification outcome, i.e. if it failed or succeeded to verify. A seed is invalid if it produces non-verification-related errors, e.g. parsing or resolution errors. We discard the seed and do not use it to generate mutants.

A valid seed is passed into the mutant generator to generate a number of equivalent mutants. Each generated mutant is passed to Dafny for verification and its verification outcome is checked against the seed's verification outcome. If there is a mismatch in the returned outcomes, we have found a potential bug and save the involved programs for further inspection. Otherwise, we continue with the fuzzing.

Assuming that the seed's verification outcome is the ground truth, there may be a soundness bug if the mutant verifies whilst the seed fails to verify. As a dual, there may be a precision bug if the mutant fails to verify whilst the seed verifies. The equivalence mutations that we apply should not

change the verification outcome of our program.

6.2 Mutant Generator

6.2.1 Usage

The mutant generator can be invoked as an executable or a library. We expose two main functionalities: `gen-single`, `gen-multiple`.

`gen-single <seed> <mutant_seed> <mutant_order>`

This functionality generates a single mutant from a `seed`. All mutants generated in our program are parameterised by a `mutant_seed` and `mutant_order`. These parameters define the combination of mutations applied to produce the mutant. By specifying these parameters, mutants can be reproduced to facilitate development and debugging.

`gen-multiple <seed> <num_mutants> <max_order>`

This functionality generates multiple mutants from a `seed` which is useful for production purposes. The number of mutants generated is specified by `num_mutants`. The maximum number of mutations that is applied to produce a mutant is specified by `max_order`.

Internally, the exact order of the mutant is chosen via a basic heuristic. If `max_order` is less than 10, we randomly choose a number between 1 and `max_order`. Otherwise, we split `max_order` into intervals which are selected using the probability distribution in Table 6.1. From the selected interval, we randomly choose an integer as the mutant order. The figures used are chosen as approximates via experimentation. We observed that mutants of very high order had a higher chance of failing or timing out. It is possible to use a more sophisticated heuristic such as mathematical distributions for order selection. We leave this for future work.

Interval (% of `max_order`)	**Probability of selecting**
0 - 10%	0.2
10 - 50%	0.5
50 - 90%	0.2
90 - 100%	0.1

Table 6.1: Selection probabilities for mutant order

6.2.2 Implementation

Overview

In the mutant generator, a series of steps is involved in generating a mutant from a seed. This is illustrated in Figure 6.2, where we present the different components within the mutant generator, and how they are chained together to form a pipeline.

We describe the steps in the pipeline. Given a seed in the form of a Dafny file, we invoke Dafny's parser and resolver to return Dafny's representation of the seed's AST that will be type-resolved. The Dafny AST is then translated into our own AST representation. To generate a mutant, we create a clone of the seed AST and mutate the clone in-place. The mutator is responsible for applying the mutations on the clone. Finally, the mutant AST is printed to a Dafny file.

We choose to implement the mutant generator in C# for interoperability with the existing Dafny toolchain also written in C#. In the following subsections, we present discussions that are specific to the components in the mutant generator.

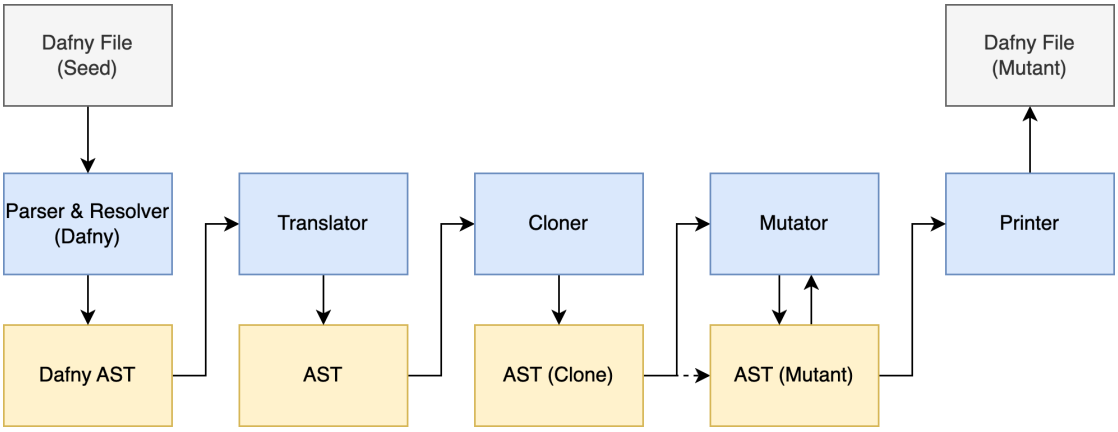

Figure 6.2: Mutant generation

Parser and Resolver

The official Dafny parser is used to process Dafny files. An alternative would be to implement a parser from scratch. We opted for the first approach as it would allow us to invoke Dafny's resolver which operates on Dafny's internal AST representation. Resolution will carry out preprocessing such as type inference that will pad the AST with helpful information.

This approach did not come for free. Working with an existing piece of complex software that does not align with the purpose of our project proved to be restrictive and a notable technical challenge. We needed to understand the complex internal workings of an advanced compiler, so that we know where and how to extract the correct information amongst the tangle of data. Additionally, some information from the original program is irrecoverable as Dafny may rewrite parts of the AST.

Translator

For this project, we look into mutating ASTs which will involve frequent node creation, node updates etc. We believe that for this context, Dafny's representation would be unsuitable to work with. Firstly, there are many references between different parts of the Dafny AST. It would be challenging to ensure all locations are updated correctly when applying mutations. Second, we observe that the procedure to construct new AST nodes is not simple within Dafny. Finally, there is a notable portion in Dafny's representation that is immutable.

Hence, we decide to implement (1) our own representation of the AST, and (2) a translator from Dafny's representation to ours. The translator will remove information that is not necessary for our purpose and create a mutable AST fit for our purpose.

Cloner

The decision for implementing an AST cloner was motivated by the following reasons. Firstly, having an AST cloner enables us to generate multiple mutants from a single seed AST. We do not have to repeat the costly stages of parsing, resolving and translation. Additionally, the cloner can be reused during mutation to produce AST components where needed.

Mutator

We define a mutator interface with a function `TryApplyMutation(Program)`. This function returns true if a mutator successfully finds and applies a mutation, and false otherwise. Our mutant generator can be easily extended with more mutators by generating classes that implement this interface.

- **Basic Mutator**

For each mutation described in Chapter 7, we implement a `BasicMutator`. These are mutators that handle a single type of mutation. Although there are no restrictions to how these mutators are implemented, we observe that the process of `TryApplyMutation` tends to follow a similar pattern illustrated as pseudocode in Figure 6.3.

```
1 BasicMutator::TryApplyMutation(program):
2   mutations := ScanForMutations(program)
3   if mutations.count == 0:
4     return false
5   mutation := SelectOne(mutations)
6   tasks := FindTasksForMutation(mutation, program)
7   foreach task in tasks:
8     task.execute()
9   return true
```

Figure 6.3: `TryApplyMutation` for `BasicMutator`

First, the program will be scanned to identify potential mutations to apply. If no mutations are found, `TryApplyMutation` will return false. Otherwise, a mutation will be selected. To apply the mutation, we identify and make a note of the changes in the AST that need to take place. The changes are applied after the list of changes to be made are determined.

We present some precautionary steps taken to ensure that the AST will be modified safely and correctly. Firstly, we should avoid mutating the AST while traversing it. Hence, we choose to separate the identification and the execution of the AST changes to apply as identification will often involve a search within the AST. Second, because identification and execution happens separately, we need to be careful that the data we collect during identification does not become stale as the execution stage proceeds. Hence, we don't try to do any work such as creating nodes or preparing references ahead of time. Work is done only when needed and always uses the up to date AST. Third, we enforce an ordering between the changes made to the AST. This helps us reason about the correctness of updates and avoid issues such as lost updates. We always choose to make changes in child nodes before parent nodes.

- **Composed Mutator**

In addition to the `BasicMutator`s implemented for each type of mutation, we implement a `ComposedMutator` which stores a list of `BasicMutator`s and applies one of them at random. This allows us to apply a mix of different mutations on a seed.

Figure 6.4 presents the implementation of the composed mutator. At each call to `TryApplyMutation`, we shuffle the list of basic mutators. Then, we apply each basic mutator in order until the first successful mutation is applied. Shuffling injects randomness in the selection of mutations. Iterating over the list ensures that each mutator is only tried once in each round of mutation so that we don't keep trying to use ineffective mutators. Reaching the end of the list means that no mutations can be found in the AST and we can stop.

```
ComposedMutator::TryApplyMutation(program):
  mutators := [basic_mutator_1, ..., basic_mutator_n]
  mutators.Shuffle()
  foreach mutator in mutators:
    if mutator.TryApplyMutation(program):
      return true
  return false
```

Figure 6.4: `TryApplyMutation` for `ComposedMutator`

Chapter 7

Implementation - Mutations

This chapter describes the set of equivalence mutations implemented in our mutant generator.

7.1 Loop Mutations

We explore equivalent ways of expressing a loop in this class of mutations. This is motivated by the lack of fuzzing for Dafny that targeted loops. XDsmith - previously the sole existing fuzzer for Dafny only generated programs that were loop-free.

7.1.1 [for-to-while] For Loop to While Loop Mutation

This mutation rewrites for-loop statements (Section 2.5.5) as while-loop statements (Section 2.5.4) As an example, Figure 7.1b shows the resultant program after applying this mutation on the seed presented in Figure 7.1a. We appreciate the simplicity of this mutation. As this was the first mutation we decided to implement, it had more use in helping us scout out and build the foundation of our fuzzer than to present itself as a fully fledged effective mutation.

```
for i := 0 to n {
  DoWork(i);
}
```

(a) Original

```
int i := 0;        // initialise loop-index to loop-start
while i != n {     // terminate upon loop-end
  DoWork(i);
  i := i + 1;      // update loop-index
}
```

(b) Mutant

Figure 7.1: For loop to while loop mutation

Applying the Mutation

We first identify a for-loop as the target for the mutation. This is done by scanning the program for all for-loops and then selecting one at random. We proceed to derive a while-loop from the selected for-loop.

Observe that the main difference between the for-loop and while-loop is the loop-index that is automatically tracked and updated. A while-loop equivalent to the target can be constructed by

the following steps. Initially, the foundation of the while-loop is formed by cloning the body and specifications from the for-loop. Then, to replicate the built-in loop index, we declare a variable `i` initialised to loop-start before the loop. Recall that loop-start and loop-end refers to the initial and final values of a for-loop's loop index. This is 0 and n for the example in Figure 7.1. Next, in the while-loop body, references to the old loop index are updated to point to `i` and an update statement is added for `i`. The update is either an increment or decrement operation via `i := i +/- 1` depending on if the loop is increasing or decreasing. Finally, we set the guard of the while-loop to `i != loop-end` to mirror the termination of the for-loop.

To complete the mutation, we replace the for-loop with the variable declaration and while-loop in the for-loop's parent.

7.1.2 [loop-peel] While Loop Peeling Mutation

Loop peeling involves extracting and executing an iteration in an if-statement out of the loop. The remaining loop is preserved and follows after the peeled iteration.

Applying the Mutation

Similar to the for-to-while mutation, selection of the while-loop to peel is a random selection over all while-loops in the program. The peeled iteration is created by constructing an if statement where the guard and body are clones of the loop guard and body.

Handling loop specifications

Loop specifications include decrease clauses, modifies clauses and the invariants. For the peeled iteration, we only consider loop invariants. We describe our reasoning behind this choice and how we replicate the invariants in the peeled iteration.

Consider each class of loop specification in turn:

- **Decreases**.
  This clause serves to provide information that enables the proving of loop termination. This is not applicable to the peeled iteration.

- **Modifies**.
  This clause serves to provide an indication of what elements may be modified as this is not easily computed in loops. For a single iteration, this is not necessary as Dafny should be capable of inferring what is modified in a non-loop construct.

- **Invariants**.
  These are expressions checked upon every loop test. They help Dafny reason about the program by providing hints on important properties about the program state. Although there may be cases where Dafny can reason correctly on its own, we choose to replicate the check for invariants to avoid false positives.

To replicate the check for invariants that happens right before and right after every loop iteration, we generate an assertion for each invariant and add them immediately before the if-statement. Assertions don't need to be added after the if statement as the check will take place in the remaining loop that follows.

We demonstrate the workings of the algorithm discussed thus far with an example. Figure 7.2b shows the resultant program after applying this mutation on the seed in Figure 7.2a.

```
1  method FindIndex(s: string, c: char)
2  returns (i: int) {
3    i := 0;
4    while i < |s| && s[i] != c      // loop to peel
5      invariant NotFound(s[:i], c)
6    {
7      i := i + 1;
8    }
9    i := if i == |s| then -1 else i
10 }
```

(a) Original

```
1  method FindIndex(s: string, c: char)
2  returns (i: int) {
3    i := 0;
4    assert NotFound(s[:i], c);      // invariant as assert
5    if i < |s| && s[i] != c {       // peeled iteration
6      i := i + 1;
7    }
8    while i < |s| && s[i] != c      // remaining loop
9      invariant NotFound(s[:i], c)
10   {
11     i := i + 1;
12   }
13   i := if i == |s| then -1 else i
14 }
```

(b) Mutant

Figure 7.2: While loop peeling mutation

Handling Breaks

Breaks with labels are not affected by the presence of a loop, hence they will not need to be modified. However, breaks without labels are interpreted relative to loops. In the extracted iteration, such breaks will need to be corrected as the loop context is modified. The effect of a break statement can vary due to potential loop nesting. There are three cases to consider.

- **Case 1: break-inside**

A break statement may not break the loop selected for peeling - if it breaks an inner loop. This is illustrated in Figure 7.3a. Assume that the outer loop on line 2 is selected for peeling. The break statement on line 5 will have no effect on the outer loop. In this case, we do not need to make any changes to the break statement in the peeled iteration. See the unmodified break on line 5 in Figure 7.3b presenting the resulting mutant.

- **Case 2: break-outside**

A break statement may break outside the loop selected for peeling - if it breaks an outer loop. This is illustrated in Figure 7.4a. Assume that the inner loop on line 3 is selected for peeling. The break statement on line 5 will break both loops. Control flow will jump outside of the outer loop. To achieve this effect in the peeled iteration, we reduce the number of breaks by one in the break statement. See the modified break on line 5 in Figure 7.4b presenting the resulting mutant.

```
1 // outer loop to peel
2 while outerCond {
3   while innerCond {
4     // breaks inner loop
5     break;
6   }
7 }
```

(a) Original

```
1  // peeled outer loop iteration
2  if outerCond {
3    while innerCond {
4      // breaks inner loop
5      break;
6    }
7  }
8  while outerCond {
9    while innerCond {
10     break;
11   }
12 }
```

(b) Mutant

Figure 7.3: Handling break-inside

```
1 while outerCond {
2   // inner loop to peel
3   while innerCond {
4     // breaks outer loop
5     break break;
6   }
7 }
```

(a) Original

```
1  while outerCond {
2    // peeled inner loop iteration
3    if innerCond {
4      // breaks outer loop
5      break;
6    }
7    while innerCond {
8      break break;
9    }
10 }
```

(b) Mutant

Figure 7.4: Handling break-outside

```
1 while cond {
2   ... break; ...
3 }
```

(a) Original

```
1  var break_loop := false;
2  label iteration:
3  if cond {
4    ...
5    // remember break
6    break_loop := true;
7    // break
8    break iteration;
9    ...
10 }
11 // check for breaks
12 if !break_loop {
13   while cond {
14   ... break; ...
15   }
16 }
```

(b) Mutant

Figure 7.5: Handling break-exact

- **Case 3: break-exact**

A break statement may break exactly the loop selected for peeling. This is illustrated in Figure 7.5a. Assume that the loop on line 1 is selected for peeling. The break statement on line 2 breaks

this loop. No further loops are broken. In this case, we need to (1) replicate the control flow jump out of the iteration, and (2) ensure that the remaining loop is not executed. This is achieved by introducing the following changes to the program.

- iteration-label: A label attached to the peeled iteration for enabling jumps out of the body.
- break-tracker: A boolean variable for tracking if a break was encountered.

Whenever a break-exact is encountered in the peeled iteration, it will be replaced by a break targeted at the iteration-label. This ends the iteration, achieving (1). Additionally, the break-tracker will be set to true. The remaining loop will be wrapped in a conditional guard that checks if the break-tracker has been set. This stops further executions of the loop if we did exit the previous iteration via a break, achieving (2). The resulting mutant is presented in Figure 7.5b.

The use of an iteration-label and break-tracker to preserve loop semantics was chosen due to its simplicity compared to other approaches. For example, we considered the scenario where statements following the break would be executed conditionally via a check to the break-tracker. This would remove the need for the iteration-label. However, it is more technically complex. We demonstrate a quick sketch of a possible scenario in Figure 7.6. For each break statement, we need to determine its successors and wrap them in a check for the break.

```
1 if x { break; }
2 work();
3 if y { break; }
4 work();
```

(a) Original

```
1  if x { break := true; }
2  // add check for break on line 1
3  if !break {
4    work();
5    if (y) { break := true; }
6    // add check for break on line 5
7    if !break {
8      work();
9    }
10 }
```

(b) Mutant

Figure 7.6: Handling break-exact - an alternative

Handling Continues

Continues are handled in the same way as breaks except for the removal of the boolean tracker in the case of a continue-exact. Unlike break-exact, a continue-exact should allow the remaining loop to continue executing, which means continues do not need to be recorded. A break on the iteration-label that ends the current iteration will be sufficient. Figure 7.7b demonstrates this case. It presents the resulting mutant originating from 7.7a.

```
1 while cond {
2   ... continue; ...
3 }
```

(a) Original

```
1  label iteration:
2  if cond {
3    // break one iteration
4    break iteration;
5  }
6  // continue further iterations
7  while cond {
8    ... continue; ...
9  }
```

(b) Mutant

Figure 7.7: Handling continue-exact

7.2 Variable Restructuring Mutations

In this class of mutations, we explore ways of expressing variables using different Dafny constructs. Specifically, we implement two mutations: one that merges variables into a map - variables are expressed as map elements, one that merges variables into a class - variables are expressed as object fields.

Throughout this section, the term "merge set" will be used to refer to the variables to be combined.

7.2.1 Motivation

It is easy for programmers to manually write a simple test case and check that it matches the intended behaviour. However, it is not possible for the programmer to write all the equivalent ways of expressing the scenario in the test case. Dafny implements many features and it is hard to ensure that the implementation is correct for all the features. The intuition behind this class of mutations is to enable the programmer to express their intent in the simplest form as variables. We can then apply this mutation to create more complex structures to cross-check that the behaviour is correctly implemented across different Dafny constructs. This is inspired from our observations on previous Dafny issues in Section 5.2 where the same scenario expressed using different constructs may have different verification outcomes.

Note that the reverse approach, i.e. deconstructing structures into variables is not a feasible approach. There may be properties enforced by the given structure that deconstructing would strip away. For example, consider an array that is sorted, we may express the sorted property as a relationship in terms of the indices and values of the array elements. Deconstructing the array into variables would make this impossible as we lose the ordering between elements.

We explore the representation of variables as map elements and class fields. These two structures were chosen as they exhibit different characteristics that we believe will allow us to exercise a range of Dafny features. Maps are immutable, built-in structures, whilst classes are mutable, user-defined and heap-based structures. Maps require all elements to have the same type, whilst there is no such restriction for classes. The entries in a map are flexible whilst the entries i.e. fields of a class are fixed.

7.2.2 [vars-to-map] Merge Variables to Map Mutation

Variables of the same type and scope can be represented as a map. The following sections describe the changes to the program made via this mutation. Table 7.1 summarises how the affected constructs will be represented in the mutant.

Original		**Mutant (merge set = { v1, v2 })**
		`var m: map<string, int> := map[];`
Initialised declaration	`var v1: int := 1`	`m := m["v1" := 1]`
Uninitialised declaration	`var v2: int`	
Use	`use(v1)`	`use(m["v1"])`
Single update	`v2 := 2`	`m := m["v2" := 2]`
Parallel update	`v1, v2 := v2, v1`	`m := m + map["v1" := m["v2"], "v2" := m["v1"]]`

Table 7.1: Changes applied by vars-to-map mutation

Creating the Map

We declare a map variable at the top of the scope enclosing the merge set and initialise it to the empty map. The map will map the variable names to the variable values. Hence, the map's key type will be the string type and the value type is the common type of the variables in the merge set.

Replacing Variable Uses

Read references to the variable are replaced by a map access indexed by the variable name.

Replacing Variable Definitions

A variable definition refers to scenarios where a variable's value is updated. This includes variable assignments as well as declarations with initialisers. Declarations without initialisers have no effect and are removed.

- **Approach**

In Dafny, maps are immutable and cannot be modified in-place. Therefore, an update to a variable in a merge set will be replaced by an update to the map variable. Recall that Dafny supports parallel updates (Section 2.5.1), i.e. a single update statement may update multiple variables. We consider the following scenarios for an update statement.

Case 1: Only one variable from the merge set is present in the set of updated variables.
As illustrated in the second last row of Table 7.1, the new map is formed using a map update expression which updates the entry indexed by the variable's name to its new value.

Case 2: Multiple variables from the merge set are present in the set of updated variables.
As illustrated in the last row of Table 7.1, the new map is formed using a map merge operation between the original map and a freshly created map whose entries correspond to the updates of the merge set variables in the statement. This preserves the semantics of parallel assignment in Dafny, where all variables are updated in the same step and are isolated to changes to other variables.

- **Reasoning**

We describe our reasoning for the approach described above.

Firstly, it is not possible to use map update expressions for parallel assignment scenarios as it only allows single key updates. Chaining map update expressions are also not possible as this introduces a sequential ordering between the initially parallel updates. Consider an assignment that swaps the values of two variables via `v1, v2 := v1, v2`. One may try to represent this as `m := m[v1 := v2][v2 := v1]`. This is wrong as we now have the update to `v1` preceding the update to `v2`, leading to the loss of the original value of `v1` during the first update.

On the other hand, the map merge operation is capable of expressing both cases. However, we chose to keep the use of map update expressions in the case of single updates for variety.

Merge set candidate restriction.

Having described how map operations replace variable definitions, we discuss how it motivates our decision to exclude variables that have been assigned to by side-effecting operations from the merge set.

The arguments to the map constructs that will replace the variable definitions are required to be expressions. Dafny disallows expressions from containing side-effecting operations - these include array or object allocations, and method calls. Hence, it is not valid to translate e.g. `v := methodCall()` to `m := m[v := methodCall()]`.

A workaround would be to introduce a temporary variable to hold the value produced by the side-effecting operation, and then use the variable during the map update, e.g. `tmp := methodCall(); m := m[v := tmp]`. We decided against this approach under the prediction that the improvements would be minimal compared to the implementation effort. Instead, we opted to only consider variables which always have side-effect free assignments for the merge set.

Handling definitions with both merge-set and non-merge-set variables.

Armed with the design of the mutation so far, we discuss a final scenario that may occur while handling variable definitions. Namely when variables which belong and don't belong in the merge set are present together in the set of updated variables for a statement.

For declarations, we illustrate out approach in Figure 7.8. We split the original declaration into (1) a declaration statement with the corresponding initialisers for non-merge-set variables, (2) an assignment statement to the map variable to update the merge-set variables. Despite introducing a sequential ordering between the updates, we do not break the semantics of parallel assignment. This is because there will be no dependencies between the right hand side values which allows them to be executed in any order. We infer this from the fact that the right hand side values will (a) not use any variables on the left as they are in the midst of being declared, and (b) are side-effect free due to the restriction described in the previous section.

For assignment statements, we illustrate out approach in Figure 7.9. We replace the updates to the merge-set variables with an update to the map variable. Updates to other variables in the statement remain unmodified.

```
var x, y := 1, 2;
```

(a) Original

```
var m: <string, int> := map[]
var x := 1;
m := m["y" := 2]
```

(b) Mutant generated with merge set { x }

Figure 7.8: Handling declarations with merge-set and non-merge-set variables

```
x, y := 1, 2;
```

(a) Original

```
var m: <string, int> := map[]
x, m := 1, m["y" := 2];
```

(b) Mutant generated with merge set { x }

Figure 7.9: Handling assignments with merge-set and non-merge-set variables

7.2.3 [vars-to-class] Merge Variables to Class Mutation

Variables of the same scope can be represented as a class object. The following sections describe the changes to the program made via this mutation. Table 7.2 summarises how the affected constructs will be represented in the mutant.

Creating the Class and Object

The class declaration representing the merge set is created and added to the program. For each variable in the merge set, we create a corresponding field in the class declaration. Then, we construct an instance of the class at the top of the scope enclosing the merge set.

Replacing Variable Uses and Definitions

All references to the variable are replaced by a reference to its corresponding field of the class instance.

Original		Mutant (merge set = { v1, v2 })

		`class C {`
		`  var v1: int`
		`  var v2: int`
		`}`
		`var c := new C`

Initialised declaration	`var v1: int := 1`	`c.v1 := 1`
Uninitialised declaration	`var v2: int`	
Use	`use(v1)`	`use(c.v1)`
Update	`v1, v2 := v2, v1`	`c.v1, c.v2 := c.v2, c.v1`

Table 7.2: Changes for vars-to-class mutation

Merge Set Candidate Restriction

- **Allow variables with side-effecting assignments**

Note that unlike for the vars-to-map mutation, we allow for variables in the merge set to have side-effecting assignments. Hence, we need to reconsider the scenario when handling declarations containing variables that belong to and don't belong to the merge set. The approach in the vars-to-map mutation relied on the assumption that the right hand side expressions are side-effect free. This is now untrue, therefore we make a slight tweak to the approach. This is illustrated in Figure 7.10. We create a declaration statement without initialisers for the variables not in the merge set. The initial declaration statement is then converted to an assignment statement that updates both the merge-set and non-merge-set variables.

```
var x, y := ReturnTwoValues();
```

(a) Original

```
var c := new C;
var y;
c.x, y := ReturnTwoValues();
```

(b) Mutant generated with merge set { x }

Figure 7.10: Handling declarations with merge-set and non-merge-set variables

- **Exclude non-auto-initialisable variables**

It is non-trivial to construct an appropriate constructor. More specifically, it is challenging to determine how all the variables in the merge set can be initialised together upon the object's construction. There are several reasons attributed to this challenge.

First, there may be a logical entanglement between the declaration of variables that are in the merge set and the declarations of those that are not that makes them hard to separate. For example, consider variables `x`, `y`, `z` where `x < y < z`, with `x < y` meaning the declaration of `x` depends on the declaration of `y`. Say the merge set `{ x, z }` is selected. Initialising `x` and `z` in the constructor is not possible without the declaration of `y` in the constructor. However, extracting the declaration of `y` is invalid as it would mean `y` is not in scope in its original place.

Second, the declaration of variables in a merge set may be a large distance apart. There may be a large chunk of code executed between the declarations, which would be unreasonable or infeasible to extract into the constructor.

Hence, we don't try to make a constructor for the constructed class declarations. This means that fields will not be explicitly initialised. Recall the concept of auto-initialisable types in Dafny (Section 2.1.4). These are types that the Dafny compiler knows how to construct default values for. Fields of these types do not require explicit initialisation. On the other hand, fields of non auto-initialisable types need to be explicitly initialised.

To workaround this issue, we can either construct default values for the fields or exclude variables of non-auto-initialisable types from the merge set. We find a middle ground between both approaches. If we encounter a non-auto-initialisable type, we try to construct an auto-initialisable type. If this fails, we exclude the variable from the merge set. For object types, we make them auto-initialisable by making them nullable. For datatypes, we add a zero-argument constructor `None` which will be the default value for the datatype.

7.2.4 False Positives

While running the fuzzing workflow described in Chapter 6, we observed false positives related to updating the map or class representing the merge set in loops. See examples in Figure 7.11, 7.12.

In loops, Dafny may drop information about an element in the program if it suspects that it has been written to. For maps, this is whenever the map variable is written to. For classes, any heap-based operations e.g. method calls and modifying object fields will trigger this issue. This holds even if the method has an empty modifies clause indicating it doesn't change the heap, or if the object/field updated in the loop is disjoint to the object/field we care about. This is discussed in the loop framing section of the Dafny reference manual [2]. Due to the loss of information, the program may fail to verify. The solution is to provide the appropriate loop specifications to recover this information.

The case for maps has not been fixed at the time of writing. For classes, we implement a workaround for this issue in two steps. First, we remove any variables that have been written to in loops from the merge set. Second, we add `modifies {}` - an empty modifies clause to loops whose body contains a method call. This works under the assumption that the methods don't modify the heap. This is a fair assumption at this stage where we have only been testing with either hand-crafted programs or programs constructed from program-generators that don't exercise heap features. A more precise solution that we leave for future work would be to find the exact modifies clause for the loop by extracting the modified fields and the modifies clause of the called methods .

```
method M(n: nat) {
  var i := 0;
  while i < n
    invariant 0 <= i <= n
  {
    i := i + 1;
  }
}
```

(a) Original

```
method M(n: nat) {
  var m: map<string, int> := map[];
  m := m["i" := 0];
  // cannot prove "i" in m
  while m["i"] < n
    // needs `invariant "i" in m`
    invariant 0 <= m["i"] <= n
  {
    m := m["i" := m["i"] + 1];
  }
}
```

(b) Mutant generated with merge set { i }

Figure 7.11: False positive due to vars-to-map mutation

```
// original
method M(n: nat) {
  var x, y := 0, 1;
  while x < n {
    // only change x
    x := x + 1;
  }
  assert y == 1 // verifies
}
```

(a) Original

```
method M(n: nat) {
  var c := new C
  c.x, c.y := 0, 1
  while c.x < n
    // needs `modifies c`x`
  {
    // only change c.x
    c.x := c.x + 1
  }
  assert c.y == 1 // doesn't verify
}
```

(b) Mutant generated with merge set { x, y }

Figure 7.12: False positive due to vars-to-class mutation

7.2.5 Extensions

There are many ways to extend this class of mutations. We present a few ideas below.

- Explore other ways of restructuring variables, e.g. `datatype T = T(v1: T1, v2: T2)` or `datatype T = T1(v1: T1) | T2(v2: T2)`.
- Reuse existing declarations. Instead of constructing new declarations e.g. a class from scratch to represent a specific merge-set. We can search for declarations that are capable of representing the variables. Alternatively, the approach can be reversed, where we select an existing declaration and populate it accordingly.
- Restructuring beyond variables. We may restructure parameters of functions and methods, or potentially even user-defined type declarations e.g. merge multiple class declarations, convert a class to a datatype. These restructurings have a more widespread effect compared to the effect of restructuring variables which is intraprocedural. For instance, changing parameter declarations will change all calls to the function or method within the program. This may lead to increased effectiveness of the mutation.

7.3 [expr-to-func] Function Extraction Mutation

An expression is selected from a program and used to construct a function. The function is added to the program, and the original expression will be replaced with a call to the function.

7.3.1 Motivation

There are two main motivations for exploring this mutation.

(1) The vars-to-class mutation (Section 7.2.3) extracts variables into classes. On its own, the mutation produces a class that only consists of fields without any function or methods. Via this mutation, we explore methods that will allow us to construct instance functions from expressions that can be used to populate the classes.

(2) Inspired by a historic issue [19] where the indirection provided by a function call was required to trigger a bug, we believe this could be an effective mutation technique.

7.3.2 Constructing the Function

There are multiple ways that we can convert an expression as a function. These correspond to the different combinations of decomposing the expression and its subexpressions. For example, a

function derived from the expression `x + 1` may be one of the entries in Table 7.3.

Function	Function call replacement for expression
`function f(i: int): int { i }`	`f(x + 1)`
`function f(i: int, j: int): int { i + j }`	`f(x, 1)`
`function f(i: int): int { i + 1 }`	`f(x)`

Table 7.3: Potential functions constructed from expression `x + 1`

We implement a `MakeFunction` function that takes an expression and returns one of the function representations of the expression. The pseudocode for this function is listed in Figure 7.13. When entering `MakeFunction` for an expression, we can (Case 1: Identity) choose to look no further and take the expression as is for an argument to the function, or (Case 2: Decompose) try to compose the expression from its subexpressions in the function. For the latter case, we will recurse into the subexpressions to form a divide-and-conquer approach.

```
make_function(e):
  pick random:
  case 1 => identity(e)
  case 2 => decompose(e)
```

Figure 7.13: Pseudocode for `MakeFunction`

Case 1: Identity

Pseudocode listed in Figure 7.14. The entire expression forms an argument to the function. A parameter corresponding to the expression's type is created. The function simply returns the parameter.

```
identity(e):
  p := make_parameter(e.type)
  return { parameters := [p], body := p }
```

Figure 7.14: Pseudocode for `Identity`

Case 2: Decompose

Pseudocode listed in Figure 7.15. An expression may be composed from its subexpressions. By calling `MakeFunction` on each subexpression, we obtain a corresponding list of sub-functions that is then merged to build the function. The function parameters are all the parameters of all subfunctions. The function body is a reconstruction of the expression using the sub-function bodies. The function specification involves merging the specifications from the subexpressions and additionally adding any specification required for the expression class. The process of constructing and merging specifications are discussed further in the next section.

Creating Instance Functions

In the process of creating the function, if an identifier to a class object is encountered, we may try to replace the object identifier with a `this` expression in the function body. This constructs an instance function that can be added to the class. Our example only describes the addition of functions to class declarations, but this can be extended to any declarations with members.

```
decompose(e):
  f_1, ..., f_n := make_function(e.subexpression_1), ...,
   make_function(e.subexpression_n)
  parameters
    := f_1.parameters + ... + f_n.parameters
  body
    := make_expression(e.class, f_1.body, ..., f_n.body)
  specifications
    := merge_specifications(f_1.spec, ..., f_n.spec) +
   create_specification(e.class)
  return { parameters, body, specifications }
```

Figure 7.15: Pseudocode for `Decompose`

7.3.3 Constructing the Function Specification

The Need for Specifications

As a verification-based language, Dafny checks that the operations in a program are legal. For example, Dafny ensures that the index is present when accessing collections, and that the precondition is satisfied when calling a method or function.

When extracting expressions into a function, the context under which the expressions initially verify may be stripped away. To avoid false positives, it is necessary to recover this context in the form of specifications, more specifically via preconditions and read frames. It is usually not necessary to provide a post-condition as Dafny should be able to read the function body and infer what it needs.

We walk through the example in Figure 7.16a. A datatype value of `OptionalInt` can only access the field `v` if it was constructed via `Some`. On line 3 where the expression is inline, Dafny infers this correctly from the assignment on line 2. If we extract the expression to a function, e.g. `function GetValue(i: OptionalInt) { i.v }`, Dafny cannot prove that the value access is safe. To fix this, we need to provide a precondition `i.Some?` that ensures that `i` will have been constructed via `Some`. The correct mutant is presented in Figure 7.16b. In the case that the value access was actually unsafe, the mutant will still be equivalent to the original program as the error will be reflected in a failed check of the function's precondition.

```
datatype OptionalInt = None | Some(v: int)
var i := Some(1)
print i.v
```

(a) Original

```
datatype OptionalInt = None | Some(v: int)

function GetValue(i: OptionalInt): int
  requires i.Some? { i.v }

var i := Some(1)
print GetValue(i);
```

(b) Mutant

Figure 7.16: Function extraction mutation with specifications

Constructing Base Case Specifications

This section describes the workings of `create_specification(e.class)`. Specifications are the conditions required for the operations that appear in the function to be legal. Some classes of expressions have known forms of specifications. When we encounter such expressions, we construct the specification by substituting the appropriate arguments into a template and add it to the function. We present a summary of the specifications that we have handled in Table 7.4.

For collection accesses, we need to check that the index is present in the collection. For datatype field accesses, we check that the datatype value matches one of the datatype constructors which contains the field. For heap accesses, e.g. when reading an array element or object field, we add the array or object being read to the read frame of the function. Other specifications that are yet to be handled include checking that divisors are non-zero, and that the specifications of another function call are satisfied etc.

Expression	Precondition	Read Frame
`map[i]`	`i in m`	
`seq[i]`	`0 <= i < |seq|`	
`array[i]`	`0 <= i < array.Length`	`array`
`object.x`		`object`
`datatype.x`	`i.ConstructorWithX1? || i.ConstructorWithX2? || ...`	

Table 7.4: Specification templates for expressions

Composing Specifications for Non-Conditional Expressions

Non-conditional expressions are expressions that don't involve decision-making. This excludes if-then-else and match expressions. An example of a non-conditional expression is a binary operation. For such expressions, its specification is the union of all specifications of all its subexpressions, together with any known specification for the expression itself.

In this case, `merge_specification(f_1, ..., f_n) = f_1.spec + ... + f_n.spec`. In Figure 7.17, we demonstrate an example that computes the specification for expression `s[i + 1]` where `s: seq<int>` and `i: int`.

```
1 spec(s[i + 1])
2 = base_spec(s[i + 1]) + spec(s) + spec(i + 1)
3 = base_spec(s[i + 1]) + base_spec(s) + base_spec(i + 1) +
    spec(i) + spec(1)
4 = base_spec(s[i + 1]) + base_spec(s) + base_spec(i + 1) +
    base_spec(i) + base_spec(1)
5 = { 0 <= i + 1 < |s| } + {} + {} + {} + {}
6 = { 0 <= i + 1 < |s|  }
```

Figure 7.17: Computing the specification for `s[i + 1]`

Composing Specifications for Conditional Expressions

A conditional expression is only decomposed if there are no specifications required in any of the branches. Otherwise, it needs to be passed in via an argument to the function.

We discuss the reasons for this restriction. Any specifications required in the branches are conditional - they don't need to hold all the time, they only need to hold when the guard to the branch

holds. This cannot be easily expressed in the overall specification. Hence, we don't process the expression any further.

The expression can be decomposed if only the guard requires a specification as we know that the guard will execute unconditionally. The guard's specification is unconditional and will appear in the overall specification.

We try to provide an idea of the intricacies involved in this problem by walking through an example. Assume that we try to construct a function from the expression `if get_value then m[i] else i`. The then-branch requires the index `i` to be present in `m`. Here are a few alternatives that one may try but are actually invalid.

- Express the specification as a conditional, e.g. `requires if get_value then i in m else true`. This is fine for this case but not scalable for more complex cases.
- Delegate the unsafe operation to the caller, e.g. creating a `function f(get_value, value, i) { if get_value then value else i }`, and replacing the expression with the call `f(get_value, m[0], i)`. This is invalid as the potentially unsafe operation `m[0]` is accessed unconditionally in the call.

There are many complexities beyond this simple example. Creating the right specifications for these expressions will need to involve some logical understanding of the expression's semantics. For now, we choose to keep things simple and use such conditional expressions as arguments to the function.

A potential solution to explore would be to use assume statements in place of function specifications. Before each operation that could trigger a verifier warning, we inject an assume statement for the safety condition of the operation. Alternatively, we can add a safety wrapper for unsafe operations. We present a sketch of both ideas in Figure 7.18, 7.19.

```
function f(get_value: bool, m: map<int, int>, i: int): int {
  if get_value then assume i in m; m[i] else i
}
```

Figure 7.18: Using assume statements for potentially unsafe operations

```
function safeAccess(m: map<int, int>, i: int) {
  if i in m then m[i] else 0
}

function f(get_value: bool, m: map<int, int>, i: int): int {
  if get_value then safeAccess(m, i) else i
}
```

Figure 7.19: Using safety wrappers around potentially unsafe operations

7.3.4 Selection Strategy

Initially, the selection of the expression to extract was done by a random selection on all expressions in the program. However, upon inspecting the mutants produced, we observed that the majority of functions produced were identity functions. This was because the number of indecomposable expressions dominated the program. An expression is not decomposable due to two reasons. Firstly, due to its nature, e.g. variable identifiers which are already in their simplest form. Secondly, the decomposition for that class of expressions has not been handled in our implementation.

To create more meaningful functions, we decide to restrict the candidates for extraction to only expressions of basic types. Decomposition has been implemented for binary and unary expressions

- these expressions tend to have basic types. Hence, we believe this restriction will increase the chances for such an expression being selected and for decomposition to occur. Additionally, this avoids the issue of extracting expressions containing generic types, which will then require the construction of generic functions.

As an afterthought, it could possibly be useful to weight the candidates based on the class of expressions it belongs to. For example, we can choose to give a higher weight to the classes that decomposition has been implemented for.

7.3.5 Extensions

- Reusing existing functions instead of creating a function from scratch.
- Merging parameters of subfunctions if they identify the same element in the original expression. Currently, duplicate arguments are not handled as we visit subexpressions disjointly and combine the parameters by addition. Consider the construction of a function from the expression `s[0] + s[1] + s[2]`. Figure 7.20a is an example of a function that our implementation may produce. However, we observe that it is actually only necessary to pass in the argument `s` once. An improvement for this function that removes the duplicate parameters is presented in Figure 7.20b.

```
// constructed function
function f(s, t, u) { s[0] + t[1] + u[2] }
// function call replacement
f(s, s, s) // only necessary to pass in s once
```

(a) Parameters duplicated

```
// constructed function
function f(s) { s[0] + s[1] + s[2] }
// function call replacement
f(s) // duplicate arguments removed
```

(b) Parameters merged

Figure 7.20: Proposed improvement via parameter merging

Chapter 8

Evaluation

To assess the effectiveness of our mutant generator tool, we investigate the coverage attained on the Dafny codebase in section 8.1, and the bug-finding capabilities of the tool in section 8.2. Section 8.3 discusses the limitations of our evaluation.

8.1 Coverage

Coverage indicates how much of the Dafny codebase we exercise. Higher coverage means more parts of Dafny are reached which increases the chances of finding a bug in the codebase. We compare the coverage achieved from fuzzing Dafny by just using seed programs versus using the mutants generated from the seeds.

During our experiments, we observed that it took around 3 minutes to verify and collect coverage for each program. Additionally, observe from Figure 8.1 that coverage starts levelling off after the 10th run. The lengthy runtimes and minimal gains in coverage after the initial runs led to our decision at capping the size of our runs to around 50 programs. We split this between three program generators - XDsmith, Fuzz-D and Dafny-Verifier. Each generator constructs 17 programs to form a total of 51 seeds.

Then, a mutant is generated per seed to obtain 51 mutants. We configured the maximum order of the mutants to be 150. This figure was chosen fairly arbitrarily. We observed that it worked reasonably well during our experiments. Mutants of higher order would fail or take much longer to verify. Mutants of lower order had less meaningful mutations as the mutation search space was large and unfocused due to the nature of randomly generated programs.

Finally, we use Coverlet [21] tooling to collect the coverage from invoking Dafny's verifier against each 51 programs. For the two runs, we plot the coverage achieved on Dafny over 51 programs in Figure 8.1. Table 8.1 shows the final coverage obtained at the end of each run.

	Seeds	Mutants
Line coverage(%)	30.6	31.7
Branch coverage(%)	25.9	26.8
Method coverage(%)	34.4	35.2

Table 8.1: Coverage from fuzzing the Dafny verifier

We observe that fuzzing with our mutants achieves a higher coverage over fuzzing with seeds, even from the first program. We attribute this to the vars-to-class mutation which generates classes in our mutants. Classes are not a feature present in the seeds.

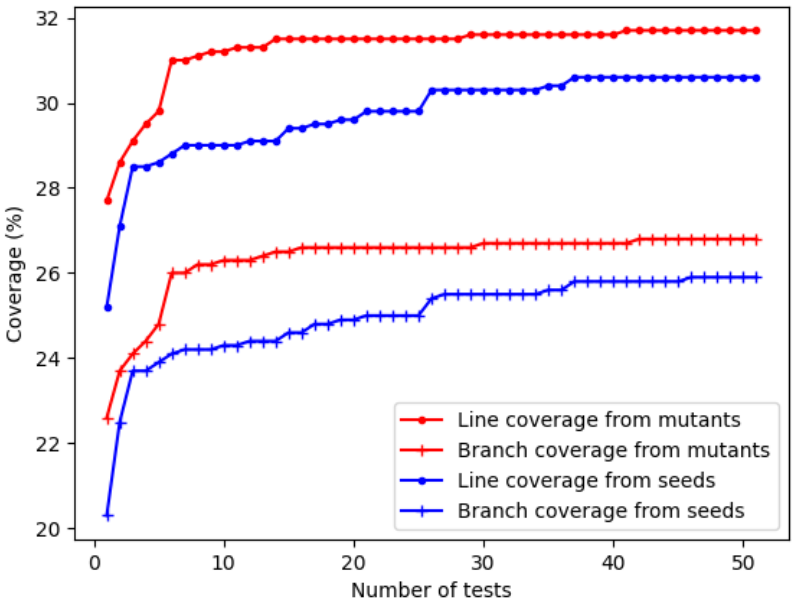

Figure 8.1: Progressive coverage from fuzzing the Dafny verifier

8.2 Finding Synthetic Bugs

Due to time constraints and technical blockers, we were unable to run a real-world fuzzing campaign. As an alternative, to demonstrate the bug-finding capabilities of our mutation techniques, we inject synthetic bugs into the Dafny codebase and fuzz it with mutants produced from hand-written programs. In these examples, we demonstrate how only simple seeds are needed to generate a bug-triggering mutant.

8.2.1 Boogie Translation Bug for If Statement

We flip the if and else branches in the translation to a Boogie program for the if statement. Figure 8.2a shows an example seed which derives the bug-triggering mutant in figure 8.2b via the loop-peel mutation.

8.2.2 Type Inference Bug for Datatype Update Expression

We recover the historic bug in [19]. It involves a type inference bug on datatype update expressions.

In the original issue, it was demonstrated that indirection by using a map and a function call was required to trigger a bug. Hence, our expectation was that both the vars-to-map and expr-to-func mutations would need to be applied to detect this bug. However, we found that the vars-to-map mutation was sufficient to detect this bug.

In Figure 8.3, the type `R` only consists of the datatype value `R(0)`. It is invalid to set the field `i` of an `R` type to a value other than `0`. An example test that checks this behaviour is presented in Figure 8.3a. This fails to verify, as expected. When a vars-to-map mutation is applied to this test, the bug-triggering test in Figure 8.3b is produced. The test verifies unexpectedly which leads us to detect the soundness bug.

```
// expect: success
// actual: success

method M(n: nat) {
  var i := 0;
  while i < n
    invariant 0 <= i <= n
  {
    i := i + 1;
  }
}
```

(a) Seed

```
// expect: success
// actual: failure

method M(n: nat) {
  var i := 0;
  assert 0 <= i && i <= n;
  // incorrect warning due to bug
  if i < n {
    i := i + 1;
  }
  while i < n
    invariant 0 <= i && i <= n
  {
    i := i + 1;
  }
}
```

(b) Bug-triggering mutant from loop-peel

Figure 8.2: Test cases for synthetic branch swap bug of if statements

```
// expect: failure
// actual: failure

type R = r: R_ | r.i == 0 witness R(0)
datatype R_ = R(i: nat)

var r: R := R(0)
// bad, correctly warns
r := r(v := 1)
```

(a) Seed

```
// expect: failure
// actual: success

type R = r: R_ | r.i == 0 witness R(0)
datatype R_ = R(i: nat)

var m: map<string, R> := map[]
m := m["r" := R(0)]
// bad, fails to warn due to bug.
m := m["r" := m["r"].(v := 1)]
```

(b) Bug-triggering mutant from vars-to-map

Figure 8.3: Test cases for synthetic type inference bug of datatype update expressions

8.3 Limitations

Our evaluation has been limited due to the time constraints and technical challenges in our project. We acknowledge them in this section and present the experiments that we would have liked to do.

8.3.1 Choice of seeds

The assessment of the effectiveness of our mutations are impacted by the seeds' characteristics.

Test suite

Ideally, we would source our seeds from Dafny's existing test suite for the following benefits. First, test cases usually target important or error-prone functionality within a piece of software. The derived mutants will inherit this characteristic, and hence be of higher value. Second, test cases are focused and easy to understand. In contrast, programs constructed via random generation techniques tend to be large programs without a coherent meaning. Therefore, mutants derived from test cases would be easier to interpret and reduce compared to mutants derived from randomly generated programs. Third, test cases and hence its mutants can exercise features that are not easily constructed via random generation techniques. For example, it is hard to automatically construct specifications such as loop invariants for a program.

However, a majority of the test cases exercise advanced features of Dafny that are not yet handled in our implementation. Thus, despite the clear advantages of sourcing seeds from Dafny's test suite, we had to fallback to using randomly generated programs in our evaluation. These randomly generated programs can be handled by our mutant generator as they tend to be restricted to a subset of Dafny features.

Randomly generated programs

The usage of randomly generated programs as seeds presented the following limitations on our evaluation.

- **Restricted Mutation Opportunities**

The seeds may not have the features that are the target of our mutations. For example, the randomly generated programs used as seeds in our evaluation do not frequently contain loops. This means that our loop mutations are less exercised and the present evaluation may not be representative of the actual effectiveness of the mutation technique.

- **Sparsely Distributed Mutations**

Randomly generated programs tend to be large, which means mutations may be sparsely distributed. This restricts our investigation into the effect of stacking mutations, i.e. applying multiple mutations to the same location.

8.3.2 Analysis of mutation techniques

It is important to evaluate the effectiveness of individual mutation techniques to understand which mutations would be worth developing further. To do this, we would choose to run our coverage experiments where only one of the mutations is enabled for all our mutations.

It will also be interesting to determine if the mutations facilitate each other, i.e. if applying a certain mutation will enhance the effectiveness of another mutation. This could be via exposing more locations possible for the mutation, or via introducing additional complexity into the program.

8.3.3 Analysis of mutant order

Running experiments with batches of mutants with different mutant orders can allow us to identify an optimal selecting strategy for mutant order. Note that this could be different for different program sizes.

8.3.4 Real-world testing

A real world fuzzing campaign would present a chance to find actual bugs and allow us to form a more real-world representative assessment of our tool.

Chapter 9

Conclusion

In this project, we investigate how to test the Dafny verifier. This paper presents an extensive study of the domain of fuzzing and Dafny, along with the rigorous reasoning about different design decisions and technical challenges that bring us to our end product of a mutation-based fuzzer armed with a set of equivalence mutations.

Additionally, we understood the difficulty of constructing specifications for Dafny programs. We appreciate the challenges of adopting a mutation-based approach where the original structure of the program may be restrictive. Furthermore, we observed how seemingly equivalent changes to a program may cause unexpected effects due to the intricate semantics of the underlying language.

9.1 Future Work

We described various points for extensions throughout this paper. These include ideas to develop our mutations further in Chapter 7 and further evaluation experiments to execute in Chapter 8. The remaining ideas are presented here.

- **Compiler Testing**
  Our fuzzer targets the Dafny verifier. However, it would be fairly simple to extend the equivalence testing to the Dafny compilers. There are multiple Dafny compiler back-ends, which indicate more potential for bugs.
- **Extend Supported Features**
  Dafny has many unique features, e.g. ghost constructs, two-state lemmas. There may be a higher potential bug for these niche features that are less explored.
- **Coverage-guided Fuzzing**
  In our tool, mutations are applied randomly on randomly selected seeds. To improve the effectiveness of our mutations, we could use coverage to guide the selection of seeds and mutations.

Bibliography

[1] Dafny;. Available from: https://github.com/dafny-lang/dafny.

[2] Dafny Reference Manual;. Available from: https://dafny.org/latest/DafnyRef/DafnyRef.

[3] Leroy X. Formal Verification of a Realistic Compiler. Commun ACM. 2009 jul;52(7):107–115. Available from: https://doi.org/10.1145/1538788.1538814.

[4] Le V, Afshari M, Su Z. Compiler Validation via Equivalence modulo Inputs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14. New York, NY, USA: Association for Computing Machinery; 2014. p. 216–226. Available from: https://doi.org/10.1145/2594291.2594334.

[5] Yang X, Chen Y, Eide E, Regehr J. Finding and Understanding Bugs in C Compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11. New York, NY, USA: Association for Computing Machinery; 2011. p. 283–294. Available from: https://doi.org/10.1145/1993498.1993532.

[6] Livinskii V, Babokin D, Regehr J. Random Testing for C and C++ Compilers with YARPGen. Proc ACM Program Lang. 2020 nov;4(OOPSLA). Available from: https://doi.org/10.1145/3428264.

[7] McKeeman WM. Differential Testing for Software. Digit Tech J. 1998;10:100-7.

[8] Klinger C, Christakis M, Wüstholz V. Differentially Testing Soundness and Precision of Program Analyzers. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2019. New York, NY, USA: Association for Computing Machinery; 2019. p. 239–250. Available from: https://doi.org/10.1145/3293882.3330553.

[9] Engler D, Chen DY, Hallem S, Chou A, Chelf B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. SOSP '01. New York, NY, USA: Association for Computing Machinery; 2001. p. 57–72. Available from: https://doi.org/10.1145/502034.502041.

[10] Chen TY, Cheung SC, Yiu SM. Metamorphic Testing: A New Approach for Generating Next Test Cases. ArXiv. 2020;abs/2002.12543.

[11] Donaldson AF, Evrard H, Lascu A, Thomson P. Automated Testing of Graphics Shader Compilers. Proc ACM Program Lang. 2017 oct;1(OOPSLA). Available from: https://doi.org/10.1145/3133917.

[12] Zhang C, Su T, Yan Y, Zhang F, Pu G, Su Z. Finding and Understanding Bugs in Software Model Checkers. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery; 2019. p. 763–773. Available from: https://doi.org/10.1145/3338906.3338932.

[13] Irfan A, Porncharoenwase S, Rakamarić Z, Rungta N, Torlak E. Testing Dafny (Experience Paper). In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2022. New York, NY, USA: Association for Computing Machinery; 2022. p. 556–567. Available from: https://doi.org/10.1145/3533767.3534382.

[14] Winterer D, Zhang C, Su Z. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. Proc ACM Program Lang. 2020 nov;4(OOPSLA). Available from: https://doi.org/10.1145/3428261.

[15] Park J, Winterer D, Zhang C, Su Z. Generative Type-Aware Mutation for Testing SMT Solvers. Proc ACM Program Lang. 2021 oct;5(OOPSLA). Available from: https://doi.org/10.1145/3485529.

[16] Le V, Sun C, Su Z. Finding deep compiler bugs via guided stochastic program mutation. Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. 2015.

[17] Sun C, Le V, Su Z. Finding compiler bugs via live code mutation; 2016. p. 849-63.

[18] Chowdhury SA, Shrestha SL, Johnson TT, Csallner C. SLEMI: Equivalence Modulo Input (EMI) Based Mutation of CPS Models for Finding Compiler Bugs in Simulink. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE); 2020. p. 335-46.

[19] GitHub Issues: Historic type inference bug for datatype updates;. Available from: https://github.com/dafny-lang/dafny/issues/1479.

[20] GitHub Issues: Historic type inference bug for map updates;. Available from: https://github.com/dafny-lang/dafny/issues/3059.

[21] Coverlet Coverage Collection;. Available from: https://github.com/coverlet-coverage/coverlet.