Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

fuzz-d: Random Program Generation for Testing Dafny

Author:
Alex Usher

Supervisor:
Prof. Alastair Donaldson

Second Marker:
Prof. Cristian Cadar

Abstract

Dafny is a modern, verification-aware programming language which enables the development of provably correct software, through writing programs annotated with specification constructs. It has become popular in teaching, academia and more recently in industry, being used to write security-critical protocols within AWS.

With this project we introduce *fuzz-d*, a compiler testing tool which randomly generates valid Dafny programs. Within *fuzz-d*, we draw inspiration from previous research to design a novel approach – *advanced reconditioning* – which avoids over-constraining program expressiveness while ensuring program validity. Additionally, we implement a Dafny interpreter to act as a reference oracle during testing, addressing limitations in applying existing testing mechanisms to Dafny. We further demonstrate that the interpreter can be used to annotate Dafny programs in verifiable ways, using it to create a workflow for testing the Dafny verifier.

Testing campaigns with *fuzz-d* have so far resulted in 14 bug reports being submitted to the Dafny developers, documenting both crashes and miscompilations across a range of language features. Of these, three have already been fixed and a further five confirmed. We have also used *fuzz-d* to identify weaknesses in the existing Dafny test suite, using both coverage analysis to find sections of code reachable by *fuzz-d* but not the integration test suite, and mutation testing to identify mutants which are killed by *fuzz-d* but survive with the integration tests.

Acknowledgements

Firstly, I would like to thank my supervisor, Prof. Alastair Donaldson, for providing invaluable insight, feedback and guidance throughout this project. It has been a pleasure to work with you!

I would also like to thank my second marker, Prof. Cristian Cadar, for his feedback on the interim report and for his teaching in the Software Reliability module, which provided a strong knowledge base for this project.

I extend my thanks to the Dafny developers for their warm reception of the project, in particular to John Tristan for offering his time to onboard us onto the Dafny codebase and sharing insights from his experience with Dafny.

Finally, to all my friends and family, thank you for your unwavering support and guidance throughout the past four years. I am privileged to have been joined by you in this journey, and could not have achieved what I have without you.

Contents

1 Introduction

1.1 Motivation

Dafny is a high-level, verification-aware programming language intended for use in building verified software. It is an extremely expressive language, providing built-in specification constructs which are used by a Floyd-Hoare-style program verifier [1] to formally prove the functional correctness of annotated programs. Consequently, Dafny has a wide range of applications in teaching and in industry-led research at large scale companies such as Microsoft [2] and Amazon [3].

Although Dafny stands out for its verifier, at its core it is a compiled language. Dafny programs are compiled into another high-level language (the *compile target language*), and this output can then be interpreted/executed using mechanisms from the target language (e.g. `dotnet` for programs compiled into C# or `node` for JavaScript). Currently, Dafny supports compilation of programs into C#, Python, Java, Go, JavaScript (mature support) and C++ (very limited support) [1].

Compilers for modern high-level programming languages are often highly complex, as a result of supporting a rich set of types, language features, and multiple levels of optimisation. This naturally creates a problem where there are a large range of possible inputs and corresponding behaviours which the compiler must be able to generate programs for. Dafny's compiler is no different – in order to be correct, it must not only be aware of the semantics of the input language (Dafny), but also the supported output languages.

Although compilers are inherently complex, they are often assumed by developers to be functionally correct and bug-free. However, as with any software application, compilers can also contain bugs. For example, the compiler testing tool Csmith has identified over 325 bugs in C compilers [4]. Almost every program that runs on a computer has been processed by a compiler or similar tool [5], and thus a bug in a production compiler can have significant consequences, especially if the bug goes unnoticed and produces an executable with invalid or incorrect behaviour. For example, several Apache applications crashed unexpectedly after the release of Java 7, which contained a miscompilation bug related to hotspot optimisation [6].

Research into compiler testing and validation has gained traction in recent years, following from the critical role of compilers and the dependence of software applications on their functional correctness. In particular, there has been a lot of focus on the area of *compiler fuzzing*, in which diverse, interesting programs are generated pseudo-randomly and subsequently used to verify the compiler's correctness. This has been applied successfully for many high-level programming languages, including C/C++ [4] and more recently JavaScript [7]. There also exists previous work on testing Dafny [3]; however, this omitted a large proportion of Dafny's language features, including loops, recursion and object-oriented data structures.

This project aims to analyse and explore existing techniques in compiler fuzzing in order to create a new tool, *fuzz-d*, which can be used to test and validate Dafny over its more complex language features. The primary aim of *fuzz-d* will be to identify bugs in the Dafny compiler, as without a reliably correct compiler implementation, Dafny's program verification would be meaningless – a written program could be formally correct, but its compiled version might not be. Equally, it is important that the verifier only accepts formally correct programs, therefore

a secondary aim of *fuzz-d* will be to identify testing mechanisms for the Dafny verifier and use these to identify bugs.

1.2 Objectives

This project aims to implement a fuzzer, *fuzz-d*, which can randomly generate programs used for testing the Dafny workflow. We undertake this project with the following primary objectives:

1. To build a fuzzer for the Dafny programming language, using existing compiler testing techniques to generate a diverse and feature-rich set of valid test programs.
2. To design novel techniques for ensuring validity of test programs and investigate their effectiveness compared to existing techniques when applied in the context of Dafny.
3. To identify mechanisms allowing for the generation of programs with a known verification outcome, using these to build a workflow for testing the Dafny verifier.
4. Analysing the ability of *fuzz-d* to test the core areas of the Dafny workflow, using these results to evaluate the effectiveness of using fuzzing for testing Dafny.

1.3 Contributions

This project makes the following key contributions:

1. We develop a novel random program generator capable of building complex, but valid, Dafny programs using a generational approach. Its design and techniques are documented in Section 4.2.
2. We implement a reconditioner module which is capable of ensuring validity of generated programs, introducing a new technique – *advanced reconditioning* – which minimises the constraint over program expressiveness introduced by reconditioning mechanisms. We also use these techniques to avoid introducing invalid behaviour during reduction of test cases. Reconditioning mechanisms are described in Section 4.4.
3. We create an interpreter for the supported subset of the Dafny language, capable of resolving limitations in performing differential testing over Dafny's backends (Section 3.4). We further use the interpreter to introduce mechanisms for annotating generated programs with valid specification constructs, integrating this into a workflow for testing the Dafny verifier. The design and techniques behind the interpreter are detailed in Section 4.3.
4. We analyse and illustrate the ability of the *fuzz-d* generator to test Dafny through:
   (a) Highlighting bugs identified by testing campaigns across multiple versions of Dafny, including both compiler crashes and miscompilations. In total, 35 bugs were found, of which there were 18 unique root causes. Four of these were previously reported, thus 14 new bug reports were submitted to Dafny developers.
   (b) Achieving 46% line coverage over the DafnyCore module of the Dafny workflow – significantly higher than the existing XDsmith tool – in a 12 hour controlled experiment.
   (c) Demonstrating that *fuzz-d* is able to both cover lines of code and kill mutants in parts of the DafnyCore module which the Dafny integration tests are unable to, highlighting possible weaknesses in Dafny's testing mechanisms.
5. We make the *fuzz-d* project available open-source on GitHub, alongside tools and resources required for the project.

2 Background

2.1 Compiler Testing

A compiler takes as input a program written in a specific high-level language, and outputs an executable version of the program in a lower-level language such as assembly or bytecode. Compilers are complex pieces of software by nature and involve interaction between several components. Figure 2.1 details the stages an input program might go through during compilation, including parsing the program, checking it for syntactic and semantic errors, applying optimisations and generating the code.

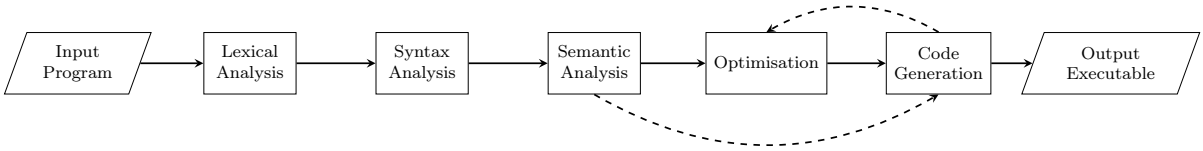


Figure 2.1: The stages of compilation

Due to the feature-rich nature of modern high-level programming languages, there is typically a broad input space of valid programs and a large number of possible paths the compilation of a program can take. Aside from core language support, compilers offer numerous additional features, such as support across different target platforms, different language versions, and several types/levels of optimisation. Optimisations can often be applied in different orders, or at different stages of compilation (e.g. before/after code generation). For example, the Clang compiler has 58 different optimisation passes [8].

The combination of a compiler's broad input space and its large number of additional features therefore makes it very difficult (if not impossible) for developers to exhaustively test all possible configurations using traditional manual testing methods. While they may be able to test more intuitive cases, it is likely that some more complex edge cases may be missed. Sometimes it is unclear what a compiler should do for particular input configurations, and this can further increase the difficulty of manual testing. For example, the C language specification does not detail the order in which optimisations should be applied.

As a result of the complexity of compilers and the importance of their correctness, a lot of effort has recently been placed into researching compiler testing, with the aim of improving compiler validity and correctness. One possible approach to solve this problem is to formally verify the compiler code, using specifications and machine-checked proofs to ensure correctness. For example, CompCert [9] is a formally verified compiler for Clight, a subset of the C language. This approach is able to eliminate the possibility of compiler-introduced bugs by formally proving that the compiler outputs executables which semantically correspond to the input program. However, such proof constructs are difficult and time consuming to create, therefore the technique does not currently scale well onto large-scale compilers – it is likely that the formal proof specification would be larger than the compiler's source.

An alternative to formal verification involves applying the automated testing technique *fuzzing* to compilers, randomly generating a large number of programs and testing these on the compiler, consequently ensuring they compile and execute with the correct behaviour [5]. The random nature of the program generation means it is likely programs will cover a large portion of the

input space, and they are also likely to be more complex (less intuitive) than manually generated test cases, thus covering possible edge cases.

Fuzzing can detect both types of compiler bugs for a particular generated program:

- *Compiler crashes* occur when the compilation process exits early without producing a valid output, typically reporting a non-zero exit code and some form of crash or error.
- *Miscompilations* occur when the compiler generates an executable, but the behaviour of this executable is semantically different from the input program. They can be harder to identify manually since the compilation succeeds without warnings or errors, thus compiler fuzzers, such as Csmith [4] and YARPGen [10], commonly identify more miscompilations than crashes.

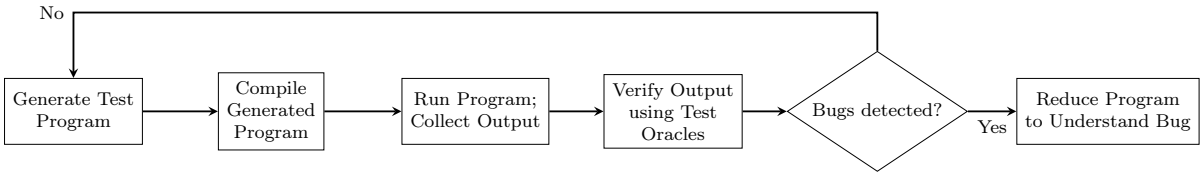


Figure 2.2: A general workflow for compiler testing using fuzzers

Figure 2.2 shows the stages which a general approach to compiler fuzzing involves. This begins with generating the test programs in the source language – techniques for generating diverse programs are detailed in Sections 2.2 and 2.3. These generated programs need to be valid in order to test the entire compilation pipeline, and this is discussed in Section 2.4. Once a program is generated, it is compiled and executed. The execution output is used to verify that the compiled program exhibits the expected behaviour – Section 2.5 discusses how we can overcome the *test oracle problem* to know the expected behaviour for a generated program. If a program is detected as containing a bug, it is *reduced* to a simpler program which can be easily debugged by a compiler developer. Methods for program reduction are discussed in Section 2.6.

2.2 Program Generation

Testing any piece of software requires *test cases* to be created, and in the case of testing a compiler, a test case is represented by a program. Constructing programs automatically, as is done in compiler fuzzing, can have many challenges. Generated programs need to be *diverse*, covering multiple different parts of a compiler in interesting ways, while being *valid* enough so as not to be rejected before the compiler stage under test. When the aim is to test the behaviour of executables produced by the compiler, a program must be able to pass through the entire compiler pipeline, thus it must be both syntactically and semantically valid. There are two common approaches to generating such programs: *generative* and *mutational* program generation.

2.2.1 Generative Program Generation

With generative program generation, test programs are generated from scratch. This can be categorised into two approaches: *grammar-directed* and *grammar-aided* [5] generation.

Grammar-Directed Program Generation

The grammar-directed approach takes as input a context-free grammar (CFG) for a language and generates test programs based on this. This is done by taking a unique start symbol and

recursively walking over the rules of the grammar in a top-down approach, applying these rules to generate strings which can be combined to form the program.

With grammar-directed generation, we can generate programs which are *syntactically correct*. Since generated programs follow the rules of the language's grammar, they can be applied to testing the lexical and syntactic analysis stages; however, the approach does not reliably generate programs that can test all parts of a compiler pipeline. This is because it is difficult to express the *context sensitive* features of a language using a CFG.

For example, given the grammar detailed in Figure 2.3, a grammar-directed generator would be able to declare and assign values to variables, but it would not have context sensitive information, for example that a variable needs to be declared before assignment or that variables can only be assigned values of their type. Therefore, it could generate programs such as `var x : Int := false` or `x := 3` (`x` undeclared) – these would be rejected by the compiler as they are not *semantically valid*.

```
statement   := declaration | assignment | ...
declaration := var <ident> : <type> := <expression>
assignment  := <ident> := <expression>
expression  := BooleanLiteral | IntLiteral
...
```

Figure 2.3: An example context-free grammar

Approaches have been suggested to provide a grammar-directed generator with context-sensitive information, such as two-level grammar techniques which involve defining an additional layer of rules/attributes. For example, Hanford proposes a system based on *affix grammars* that uses syntax generators to store information about declared labels [11], e.g. if we declare `x` then a rule for `x` is added into the CFG by the syntax generator. With this example, the generator would know a variable can only be used after declaration. However, we can see that two-level grammars are still limited in the extent to which they can provide a generator with context sensitive information and grammar-aided approaches are able to provide a more generalised solution.

Grammar-Aided Program Generation

Grammar-aided approaches also take a language grammar as input (or it may be hard coded into the generator implementation), but unlike grammar-directed approaches, they instead use heuristics to handle context sensitivity.

An example of a grammar-aided generator is Csmith [4]. After generating a number of top-level members and storing these in a *global environment*, it takes a top-down approach to generating C code, starting with the `main` function and using the grammar along with probability tables and filters to select future constructs to generate. These selection techniques, along with other complex heuristics, help Csmith to generate programs which are semantically valid and free of undefined behaviour. YARPGen [10] also takes a similar approach, except it is able to use heuristics to avoid introducing undefined behaviour or semantic errors in expressions.

2.2.2 Mutational Program Generation

Mutational program generation takes the approach of modifying parts of an existing test program, by applying a series of transformations to produce a *mutated program*. Note that it is possible the existing program was created via a generative approach. There are two possible

approaches to these transformations: *semantics-preserving* and *non-semantics-preserving* transformations. Typically, mutational approaches are used to find bugs in the optimisation phase of a compiler, since the mutations complicate control flow and are therefore likely to engage the optimiser in different ways.

Semantics Preserving Mutations

Semantics preserving mutations transform a program in ways which do not alter its runtime behaviour. This is based on the idea of *equivalence modulo inputs* (EMI), which defines two programs as equivalent under a set of inputs if they exhibit the same behaviour over that set of inputs. Le et al provide a formal definition in [12]: given programming language $\mathcal{L}$, two programs $P, Q \in \mathcal{L}$ are equivalent w.r.t. input set I iff $\forall i \in I :: [\![P]\!](i) = [\![Q]\!](i)$. When applied to compiler testing, the concept of EMI ensures that two test programs will have the same output under a set of inputs – this has applications in solving the test oracle problem and is discussed in Section 2.5.

An example of a bug-finding tool using these transformations is Orion [12], which applies the "profile and mutate" strategy to identify unexecuted sections of a program (*dead code*), then generating a number of EMI variants by randomly applying mutations to identified dead code. These mutations involve removing statements, which should have no impact on the behaviour of the original program.

While Orion is limited to pruning programs, a more powerful example is GLFuzz [13], which is applied to testing the graphics shading language OpenGL to generate equivalent shader programs. When rendered, the equivalent shader programs should be visually equivalent – a rendered shader which differs significantly from another is seen as a miscompilation bug. As well as pruning dead code, GLFuzz adds new code to unexecuted sections via generative techniques, demonstrating how generative and mutational techniques can coincide. It also leverages semantics-preserving identities to mutate numeric and boolean expressions in live sections of code, e.g. $sin(-x) = -sin(x)$.

Non-Semantics Preserving Mutations

Unlike semantics preserving mutations, non-semantics preserving mutations do not have to maintain that the mutated programs exhibit the same runtime behaviour as the original, and therefore they have much more freedom in the ways that they can transform a given program. They are used with the aim of ensuring a generated program is valid and suitable for compiler testing, for example by mutating it to avoid undefined behaviour.

Nagai et al. use this technique to avoid undefined behaviour in arithmetic expressions when generating C programs – first by shortening the length of the expressions [14], and later using heuristics [15]. These heuristics involved flipping operators of an expression to avoid overflow (e.g. from $+$ to $-$), or adding checks to ensure a denominator of a division cannot be zero. By adding the heuristics, they were able to generate a wider variety of expressions which were known to be free of undefined behaviour, thus able to detect more bugs.

2.3 Optimising Program Generation

Compilers are widely used and typically well-tested. Therefore, it can be hard to identify latent bugs present in compilers, and long periods of testing time are typically required to identify even a small amount of bugs. As a result of this, research is ongoing into ways in which this process can be optimised. In particular, we want to be able to generate test programs which are diverse and have high probability of triggering bugs.

Diversity of generated test programs is important in optimising compiler testing since there is low probability of an individual test program triggering a bug. Therefore, by covering more language features and more areas of the compiler, the testing will have more value than generating tests which all cover a similar part of the language. One approach to producing diverse test programs is Swarm Testing [16], discussed in Section 2.3.1. There is also research ongoing into using information about previously discovered bugs to determine language features which are more likely to lead to bugs (Section 2.3.2).

2.3.1 Swarm Testing

Groce et al. introduce Swarm Testing [16] as an efficient way to increase diversity of generated test programs. It is built on *configuration-based testing*, and involves creating a "swarm" of different configurations corresponding to the language features used in test programs. With this idea, test programs can be generated which may deliberately omit certain language features in order to more intensely test a subset of the language.

Previous testing approaches generally showed adversity to language feature omission – they would typically opt for a single "optimal" configuration, meaning an omitted language feature would remain untested. Swarm testing overcomes this by using a range of randomly generated configurations, with which it can typically produce a more diverse set of test progams than previous approaches. Furthermore, some bugs are more likely to be found when using a smaller subset of language features. Groce et al. [16] give the example of a stack with push and pop operations – if there's a capacity bug which only occurs when the stack has size 32, then we need to have a minimum of 32 push operations. If each stack operation had equal probability of being selected, then the probability to generate a test case triggering the bug would be $\frac{1}{2^{32}}$, but if we used swarm testing to omit the pop operation, then the probability of triggering the bug increases substantially.

2.3.2 AI-based Approaches

Recently, program generators using artificial intelligence-based approaches have been a popular topic of research. They are generally implemented over a mutational fuzzing approach with the aim of analysing the ability of AI to improve program diversity, commonly by optimising compiler coverage or identifying language features which have a higher probability of triggering a bug.

The evolutionary technique *genetic programming* can be used to target generation of uncommon code fragments. IFuzzer [17] uses this approach to test SpiderMonkey, the Mozilla JavaScript interpreter, taking as input a corpus of test programs and breaking these down into a pool of fragments, from which it forms complete programs and evaluates them using the interpreter. While genetic programming has had some success, *neural network language models* (NNLMs) have been demonstrated to identify significantly more bugs. For example, Montage [7] uses an NNLM to convert JavaScript test suites into fragment sequences, in order to learn the compositional relationships between them. The trained model selects fragments which are the best fit for a given AST mutation context and inserts these to form new test cases.

Reinforcement learning techniques have also been investigated for optimising generation. This involves an agent interacting with an environment to learn the best ways to optimise mutations of an AST state for compiler coverage, through inserting/replacing/deleting code fragments. While fuzzers using the approach, such as FuzzBoost [18], have achieved good compiler coverage, there is little evidence of it being able to detect bugs effectively.

2.4 Validity of Test Programs

Although there are a wide range of possible inputs for a compiler, a large number of these are *invalid* as they do not meet the syntactic or semantic requirements of the language. While it would be possible to test a compiler front-end – parsing and syntax checking – using invalid inputs, they would not be effective for checking for compiler miscompilation, for example. Therefore, in order to test the entire compiler pipeline, an effective test program should be syntactically and semantically valid, meeting the constraints imposed by the language. They need to execute in a deterministic manner such that the produced output is meaningful and easily reproducible.

While it is relatively straightforward to avoid generating programs with syntax errors, it can be a lot less trivial to avoid semantic errors, particularly implicit semantic constraints which the compiler does not check for, but which cause errors or issues at runtime. In some languages such as C, such runtime-level semantic errors are often cases of *undefined behaviour*, which refers to program behaviour that has no explicit definition in the language specification. Examples of undefined behaviour include accessing an out-of-bounds array index or dereferencing a null pointer. Compilers make no formal guarantees about the runtime behaviour of code exhibiting undefined behaviour, and are therefore allowed to generate arbitrary code for this case, making the output of a program with undefined behaviour non-deterministic.

Even if a language specification contains no undefined behaviour, an effective test program should still avoid runtime-level semantic errors as these often instead result in exceptions and cause early termination of the program, possibly preventing the program from producing a meaningful output which can be used to check its behaviour.

2.4.1 Structural Approach

An easy solution to ensuring validity would be to simply avoid generating the structures which can lead to invalid test programs. For example, we could avoid generating array accesses (or supporting array types), to ensure that programs have no out-of-bounds accesses. Quest [19] takes this approach as it only focuses on testing a small part of the C language – testing C calling conventions – and therefore has no need to worry about generating complex arithmetic expressions, for example, which could lead to integer overflow. Taking the structural approach helped Quest avoid solving unnecessary problems in its generator implementation, but this approach limits the diversity and expressiveness of the generated programs, so it would not scale well onto a larger context where a large portion of the language is being tested, as is the case with *fuzz-d*.

2.4.2 Dynamic Checks

An alternative solution ensuring program validity at runtime is to implement a series of wrapper functions. These can be called by a generated program to ensure operations with runtime-dependent values are safe and meet implicit semantic constraints. For example, we can use a wrapper function to ensure an arithmetic operation does not involve division by zero, demonstrated in figure 2.4.

This approach was taken by Csmith [4], which uses wrapper functions and other similar heuristics, such as applying modulo to array indexes, to ensure arithmetic operation safety and avoid triggering undefined behaviour. Although CSmith uses this to great success and was able to identify a large number of bugs using this technique, it applies the wrapper functions unconditionally and this was seen by Livinskii et al. [10] as heavy-handed and a limitation in the expressiveness of generated programs. Even-Mendoza et al. [20] provide empirical evidence that this may limit the bug-finding capabilities of Csmith and that it is possible to relax the restrictions introduced by Csmith while maintaining freedom from undefined behaviour. They

```
method Main() {
    var x := 5;
    ...
    var z := x / y;
}
```

```
function method safeDiv(x: int, y: int): int {
    if (y = 0) then x else x / y
}

method Main() {
    var x := 5;
    ...
    var z := safeDiv(x, y);
}
```

Figure 2.4: An example of using a wrapper function to protect against unsafe division in Dafny.

do so by instrumenting wrappers with print statements to identify required wrapper functions, removing all others to increase expressiveness.

Recent work by Lecoeur et al. [21] introduces the technique "reconditioning" which decouples program generation from the process of ensuring program validity. Whereas Csmith [4] annotates generated programs with safety wrappers at generation-time, reconditioning instead transforms this into a "just-in-time transformation" [21], done just before a generated program is tested. Using reconditioning provides greater flexibility as it ensures that we always maintain validity of a test program, even during test case reduction (Section 2.6). Figure 2.5 shows a workflow for fuzzing using the reconditioning method.

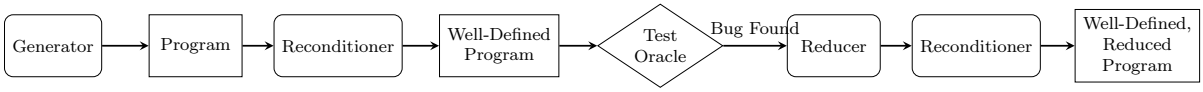


Figure 2.5: A fuzzing workflow using reconditioning (inspired by [21])

While reconditioning is able to ensure validity of programs, it still does not overcome the limitations that safety wrappers impose upon the expressiveness of programs.

2.4.3 Generation-time Static Analysis

While dynamic analysis places checks on values at runtime to meet implicit semantic constraints, we can instead use information available at generation time to ensure that constructs and values being generated are always valid, avoiding the need for dynamic wrapper functions. This constitutes a form of static analysis, where the generator tracks the values of expressions (most notably, identifiers) so it can ensure safety of generated arithmetic operations. For example, if we want to generate `x / y` but know that `y = 0`, we can change this to `x / (y + 1)` to ensure safety.

YARPGen [10] takes this approach to generating test programs, interleaving between code generation and analysis to efficiently generate valid code, without the expressiveness limitations of CSmith. While static analysis is able to avoid redundant safety checks, it quickly becomes a very complex and hard-to-solve problem, and this is demonstrated in YARPGen supporting a much smaller portion of the C language than CSmith, notably without support for function calls and classes (C++). Introducing support for function generation becomes increasingly difficult as modelling the values of function parameters is a problem related to *symbolic execution*.

Comparing Program Validity Methods

Selecting a solution to the problem of test program validity marks a trade-off between implementation complexity and language expressiveness. For *fuzz-d*, it would be too complex to avoid using any dynamic checks for the desired language features (classes, function calls, loops, heap structures etc.), but where possible *fuzz-d* should avoid using redundant dynamic checks so as not to obscure bug detection with unnecessary additional code. Therefore, *fuzz-d* will take an

approach which uses dynamic checks to meet the implicit semantic constraints of Dafny, but with the option of using advanced reconditioning to ensure that only necessary dynamic checks are present in the final test case.

2.5 Test Oracles

The concept of the *test oracle* was introduced by Howden [22] and represents a way to check a test's output to determine if it is correct – a common example would be checking output values are correct over a set of input values. With traditional manual testing approaches, a developer is usually able to determine the expected behaviour and encode it into the test case. However, for automated test cases this process is less trivial and introduces the *test oracle problem*, encapsulating the idea of detecting/inferring a test's output for a given input. There are two approaches which can be taken towards solving the test oracle problem for testing compilers: *differential testing* and *metamorphic testing*.

2.5.1 Differential Testing

McKeeman defines differential testing as a way to minimise the cost of evaluating test results [23]. It involves taking at least two comparable programs, and comparing their results over the same input. If one of the programs exhibits a different output, then a bug may have been found.

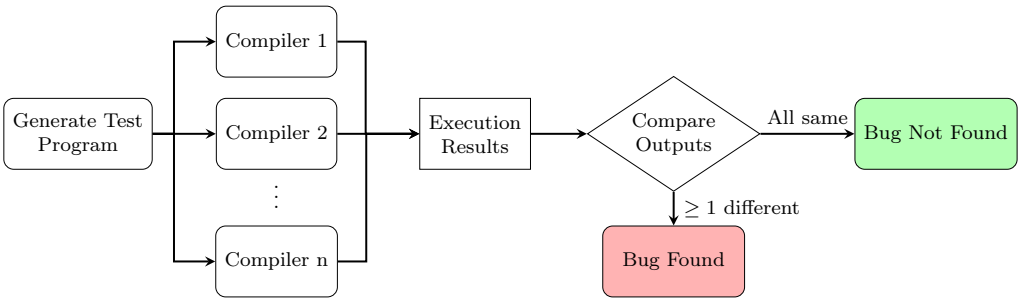


Figure 2.6: Workflow for fuzzing with differential testing

Figure 2.6 shows how this can be applied to compiler testing: a program is compiled in a number of ways, the executables are run and outputs collected, and then the outputs are compared to identify any differences. There are a number of variants in how the programs are compiled:

- *Cross-compiler* – This is the most general version of differential testing on compilers, involving compiling a generated program using multiple different compiler implementations, e.g. using both Clang and GCC compilers for a C program. For this, multiple compilers need to exist that are implemented using the same specification and language version. This limits its applications to older, more popular programming languages as it is likely newer programming languages will only have a single compiler.
- *Cross-optimisation* – A program is compiled using a single compiler across a number of optimisation levels that are implemented within the compiler. This is the most widely used strategy within compiler testing, but requires a compiler to implement flag-enabled optimisations.
- *Cross-Version* – The program is compiled using different versions of the same compiler. It is useful as it can help detect regressions between releases – bugs can be introduced between compiler versions.
- *Cross-Scenario* and *Cross-Architecture* are less common and more case-specific versions of differential testing using a single compiler. Cross-scenario focuses on obtaining results

across different compilation scenarios, e.g. just-in-time compilation vs. compilation into a machine-independent intermediate language. Cross-architecture compares results across executables for the different architectures supported by the compiler, e.g. x86 vs ARM.

2.5.2 Metamorphic Testing

Metamorphic testing [24] involves taking a test program and using it to construct *metamorphic relations*, describing how changes to the test program would affect its output. When applied to compiler testing, instead of using multiple compilers as with differential testing, a program is taken and metamorphic relations are used to construct a range of different programs whose output we can analyse.

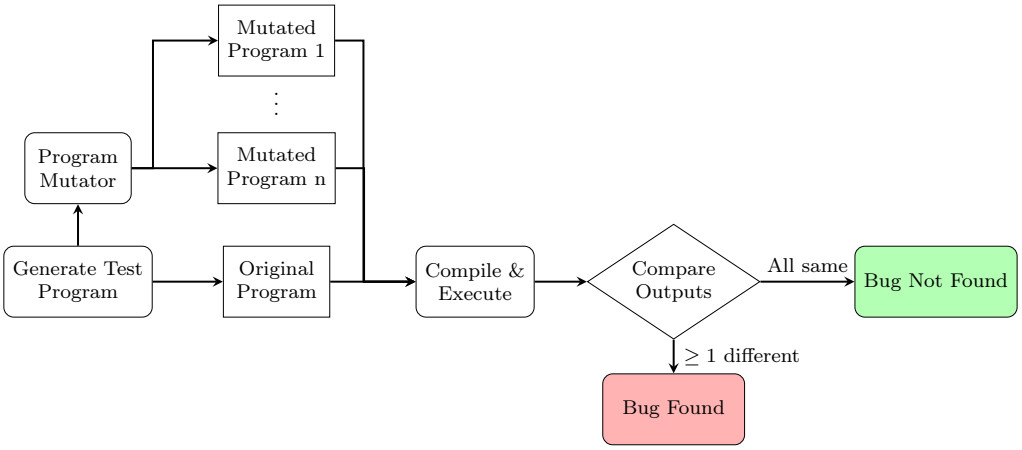


Figure 2.7: Workflow for fuzzing with metamorphic testing

The most popular method used to construct metamorphic relations for compiler testing is using equivalence relations, which result in programs that exhibit the same behaviour (thus are equivalent) after mutation. Equivalence modulo inputs (EMI) testing [12] builds upon this idea, using semantically equivalent transformations to generate mutated versions of the test program which produce the same output on execution. Common examples of equivalence relations include inserting/removing code from unexecuted sections (dead code), or altering arithmetic and boolean expressions using identities, e.g. we could replace `x` with `true ? x : y` [13].

Figure 2.7 details a workflow for compiler testing: a test program is generated and passed to a mutator which uses equivalence relations (e.g. EMI) to produce mutated programs, whose outputs are compared to identify any bugs. We can detect a bug via any difference in output since the equivalence relations result in programs which should produce the same output.

2.6 Test Case Reduction

Once a bug has been identified using compiler testing, it can be submitted to compiler developers so it can be fixed. However, since fuzzing generate programs randomly, interesting (buggy) programs are often long and complex, thus it would be hard for developers to read the program and understand where the issue lies. For example, Figure 2.8 displays an example code snippet generated by the *fuzz-d* tool (presented in Chapter 4). It would be neither helpful nor appreciated by the developers to submit such a complex program in a bug report.

Therefore, before a bug can be reported, the relevant test program should be *reduced* in order to make it easier for a compiler developer to find the problem – the guidelines for submitting a bug report for LLVM require that a test case is reduced [25]. For this, we want to identify the smallest program possible which still exhibits the unexpected behaviour. A common solution

```
match (if (v0) then v58 else fm2(v54.f8, 0x42ac, globalState)) {
    case DC60(cf99, cf100) ⇒
        globalState.f19 := (cf100 − −|v38|) + (−cf100 − v54.f108);
        var v59 := DC128(fm0(v54.f8, v36[safeIndex(v2, |v36|)], v54.f8,
            globalState));
        v54.m3(v54.f8 = |{v54.f8, v54.f8}|, v59.cf198, cf100, globalState);
        globalState.f11 := −safeDivisionInt(v54.f8, cf100) + |"nyd"|;
    ...
}
```

Figure 2.8: An example Dafny program generated using *fuzz-d*

to this problem is *test program reduction* which takes in the test program as input, along with some interesting behaviour it should observe (i.e. a way to check for the bug), and then tries to both simplify and shorten the program while ensuring it still contains the bug.

There exist a number of custom programs for automatically reducing a test program into a simpler form. These can be implemented specifically for a particular language or domain, such as glsl-reduce [26] which reduces programs in the GLSL language. Seq-Reduce [27] is another example of a domain-specific reducer. It requires interaction with Csmith to run, using additional internal modes and bypassing its pseudo-random number generator (PRNG) to randomly modify and reduce programs.

Alternatively, automatic program reducers can be implemented in a more generic, language-agnostic fashion. C-Reduce [27] takes a more generic approach to test case reduction, invoking a series of "pluggable transformations" to reduce a given program until a global fixpoint is reached. These transformations, such as removing unused functions/variables and inlining short functions, aren't specific to the C language and could easily be used for reducing programs in other languages. Similarly, PERSES [28] takes a language-agnostic approach by applying more generic transformations onto an internal AST representation. Whereas C-Reduce can enable some reduce transformations specific to C/C++, PERSES can reduce programs in an arbitrary language since it takes a language grammar as part of its input.

Typically, program-reduction needs to be language-specific in order to avoid introducing undefined behaviour into reduced programs. For example, Fast-Reduce [27] uses domain-specific knowledge to avoid reducing C programs in such a way that invokes undefined behaviour. However, *fuzz-d* will instead take a language-agnostic approach to test case reduction, utilising a reconditioning mechanism to ensure that reduced programs do not introduce invalid program behaviour.

Bug Slippage

In both language-specific and language-agnostic test case reduction, it is possible for *bug slippage* to occur, where reducing a program with one type of bug, α, could lead to it exhibiting another bug, β. This is undesirable as β may have already been identified and the test case reduction could mask the discovery of new bug α. In order to mitigate the harmful effects of slippage and harness it in a way which can be used to find more bugs, Holmes et al. [29] introduce the idea of producing a set of reduced test programs, rather than a single result.

3 The Dafny Language

Dafny is a verification-aware programming language created at Microsoft [30], and is currently being used and researched within AWS [3, 31]. Its language is imperative in nature and of very similar structure to other modern high-level languages, particularly those it compiles into. Consequently, there are a lot of common language features which Dafny shares with other languages, but it does also implement some more unique features which are less common, taking inspiration from mathematical constructs and functional programming.

This chapter aims to describe the Dafny compiler workflow and introduce both the core and verification-oriented features of Dafny which are implemented in *fuzz-d*, providing examples of use in practice for those which are less common and more unique to Dafny. We also describe some implicit semantic constraints which are not checked for by the Dafny compiler, but which may cause runtime exceptions.

3.1 Dafny Compiler Workflow

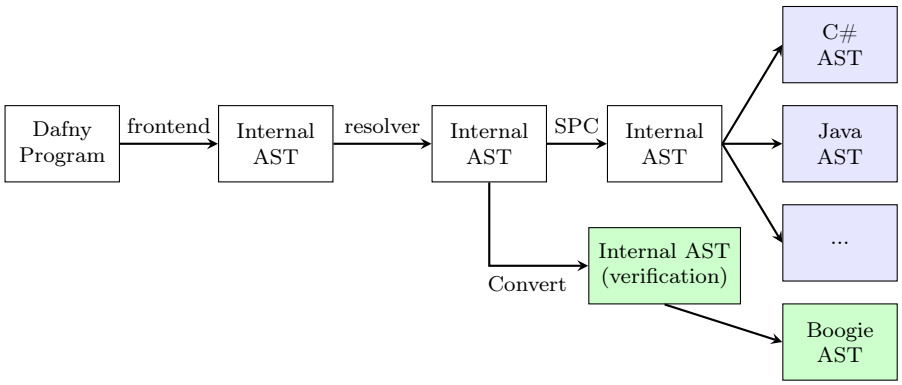


Figure 3.1: Dafny Compiler Workflow, inspired by [32]

The Dafny compiler workflow is shown in Figure 3.1. A program first passes through the compiler front-end and resolver (syntactic, semantic and type checks), resulting in an internal representation of the program in the form of an *abstract syntax tree* (AST). After this, the workflow splits and can be considered as two separate paths – one for verification and one for compilation. The verification path transforms the internal AST from the resolver into one for use in the verifier, which is then converted into an internal verification language (Boogie [33]) and passed to a satisfiability modulo theories (SMT) solver. For the compilation path, a *single pass compiler* (SPC) first transforms the resolver's AST into a common internal AST for compilation, before being converted into an AST targeted towards the specific output language – "pretty printing" this final AST produces the output program by converting each AST construct into its corresponding language representation.

3.2 Core Language Features

3.2.1 Primitive Types

Dafny supports common primitive types, including booleans (`bool`), integers (`int`), characters (`char`) and floating point numbers (`real`). It also supports a subset type for natural numbers (`nat`), which represents the non-negative range of integers. Almost all arithmetic and logical operators are supported over primitive types, notably extending the commonly supported logical operators to include equivalence (`<==>`), implication (`==>`) and reverse implication (`<==`).

3.2.2 Value and Collection Types

Alongside primitive types, Dafny features a number of built-in collection types: sets, sequences (ordered lists), strings, multisets and maps. In Dafny, a collection type represents a type which stores information that does not depend on the state of the heap, and cannot be modified once created – operations on them will result in a new copy of the collection. This distinction is made against similar high-level programming languages, where a collection may be mutable and dependent on values in the heap, so that the types can easily be used in specifications as well as compiled code.

All collection types in Dafny support element membership, selection and element updates. Most support a concept of union, difference and intersection, as well as comparison operations e.g. subset (`<=`) and superset (`>=`). The functional-inspired feature of comprehensions is implemented in Dafny for the creation of maps, sets and sequences, where a range and function are provided for use in generating the elements. For example, Figure 3.2 uses a comprehension to create a map for the numbers 0 to 99, storing whether or not they are even.

```
1 method Main() {
2     var m := map i | 0 ≤ i < 100 :: i := i % 2 = 0;
3 }
```

Figure 3.2: An example of instantiating a map using a comprehension.

3.2.3 User-Defined Datatypes

In Dafny, users can define two kinds of algebraic datatypes: *inductive* and *co-inductive* datatypes. The difference between these is that Dafny evaluates constructors for co-inductive datatypes lazily, allowing infinite structures. Both inductive and co-inductive datatypes are defined by a list of constructors for the datatype, for example `Nil` and `Node` in the definition of a binary tree in Figure 3.3. To make this a co-inductive datatype, we can replace `datatype` with `codatatype`. We don't necessarily have to pass parameters into the datatype constructors, and this can be used to achieve *enum* types e.g. `datatype Colour = Blue | Red | Green`.

```
1 datatype BinaryTree<T> = Nil | Node(left: BinaryTree<T>, value: T, right: BinaryTree<T>)
```

Figure 3.3: An example of creating a datatype for a binary tree in Dafny.

3.2.4 Reference Types

Reference types in the Dafny specification refer to objects which are dynamically allocated space in the heap upon creation. They can have multiple object members, which *dereference* the object when they are accessed. Of these reference types, *fuzz-d* supports *arrays*, *classes* and *traits*.

Arrays are supported in Dafny in a very similar fashion to many other high-level programming

languages. They can have multiple dimensions (at least 1) and be of arbitrary length, although *fuzz-d* always defines length on array initialisation for generation simplicity.

Classes form the basis of Dafny's object-oriented features. A class can have multiple fields, which can be constant or mutable, and member functions. They support multiple constructors, of which a *nameless* constructor is considered the default constructor, and can be instantiated using the `new` keyword, for example `var c := new C(args)`.

A class cannot yet extend other classes, but it can extend multiple **traits**, which behave in a similar way to a Java interface or Scala traits. They define a number of fields, functions and methods which a class extending the trait must implement, for example those extending `Shape` in Figure 3.4 must implement the function method `Area`. Traits can optionally extend other traits.

```
1  trait Shape {
2      function method Area(): real
3  }
4
5  class Rectangle extends Shape {
6      var sideLength1: real, sideLength2: real;
7
8      constructor(sideLength1: real, sideLength2: real) {
9          this.sideLength1 := sideLength1;
10         this.sideLength2 := sideLength2;
11     }
12
13     function method Area(): real {
14         sideLength1 * sideLength2
15     }
16 }
17
18 method Main() {
19     var rect := new Rectangle(10.0, 12.0);
20 }
```

Figure 3.4: Example usage of classes and traits in Dafny.

3.2.5 Control Flow: Loops

```
1  method Main() {
2      var a := new int[10];
3      forall i | 0 ≤ i < a.Length { a[i] := i; }
4  }
```

Figure 3.5: Example usage of the `forall` parallel assignment construct in Dafny.

Both while loops and for-loops exist in Dafny, implemented similarly to other high-level languages. There also exists a speciall `forall` iteration construct, which can be used in executable code for parallel assignment to array or object fields, although its role is more prevalent for quantification in code annotations. Figure 3.5 shows an example of using Dafny's forall construct to assign values to an array.

3.2.6 Methods vs. Functions

Whereas most high-level languages only have one type of function, Dafny distinguishes between *methods* and *functions* to make it easier for the verifier to reason about program behaviour. Methods are represented by a list of statements which can freely read from and alter the program state (variable/heap values), while functions are mathematical constructs consisting of a single expression that can only read from the program state. This difference stems from Dafny's limitation on expressions – they cannot be *effectful* [1] and so are not allowed to mutate the

program state in any way. This differs from many high-level programming languages where side-effecting expressions are common, for example `x++` in C or Java.

Since methods are able to mutate the heap, this makes them potentially effectful and consequently they can only be called as a statement or as the right-hand-side of a declaration or assignment, whereas functions can be called from anywhere within an expression since they are side-effect-free.

3.3 Verification-Oriented Language Features

Verification of a Dafny program utilises *modular verification*, where verifying each independent component of a program individually implies the correctness of the complete program. Dafny therefore includes a number of *specification constructs* which are user-defined logical statements corresponding to intended program behaviour. During verification, Dafny checks that these constructs hold true over any possible program execution. This section will briefly cover some of the core specification constructs which are implemented in *fuzz-d*.

3.3.1 Pre-conditions, Post-conditions and Invariants

Pre- and Post-conditions are used in Dafny to define a methods's (or functions's) *specification*, which describes the logical constraints on a methods's parameters and return values respectively. Dafny uses these to reason about functions as Hoare triples $\vdash \{P\}\ C\ \{Q\}$, where post-conditions Q are expected to hold when C terminates, assuming pre-conditions P hold.

The keywords `requires` and `ensures` are used to represent pre- and post-conditions in Dafny. For example, in Figure 3.6, the pre-condition on line 2 constrains the length of the input array, while the post-conditions on lines 3 and 4 define that the output value is the minimum in the array.

```
1  method Min(a: array<int>) returns (min: int)
2      requires a.Length > 0;
3      ensures forall i :: 0 ≤ i < a.Length ⟹ min ≤ a[i];
4      ensures exists i | 0 ≤ i < a.Length :: min = a[i]
5  {
6      min := a[0];
7
8      var i := 1;
9      while(i < a.Length)
10         invariant 1 ≤ i ≤ a.Length;
11         invariant forall j :: 0 ≤ j < i ⟹ min ≤ a[j]
12         invariant exists j | 0 ≤ j < i :: min = a[j]
13     {
14         if(a[i] < min) {
15             min := a[i];
16         }
17         i := i + 1;
18     }
19 }
```

Figure 3.6: An example (annotated) Dafny program to calculate the minimum value in an array

Loop invariants are required in order for Dafny to be able to verify termination of a program and prove that any specification constructs after the loop hold. In Figure 3.6 the invariant on line 10 ensures termination by constraining `i`, while the invariants on lines 11 and 12 define `min` as the minimum value in the array so far (up until `i`). Thus, we can see the post-condition will hold after the loop terminates.

3.3.2 Framing Constructs

Dafny uses a specification style based on *dynamic frames* for structuring the heap. This originates from the *frame problem*, which describes the problem of knowing which parts of a system are/are not affected by a particular change [34]. Dafny uses this notion to define a dynamic frame as a set of objects in the heap.

Alongside dynamic frames, Dafny uses the idea of *footprints* to represent the set of fields a program or component is permitted to read or modify. A program or component's footprint is defined in its specification, using `reads` and `modifies` expressions. A `reads` expression is used in a function to define heap objects which can be read by the function – the verifier enforces that only those which are listed are read. Similarly, a `modifies` expression is used in a method to define heap objects which can be altered by the method body, again enforced by the verifier. For example, Figure 3.7 shows the use of `modifies` within a function that sets an array index to a given value. Without the expression on line 2, the verification fails since we could be modifying a heap structure unintentionally.

```
1  method SetValue(a: array<int>, i: int, value: int)
2      modifies a;
3      requires 0 ≤ i < a.Length;
4      ensures a[i] = value;
5      ensures forall k | 0 ≤ k < a.Length :: k ≠ i ⟹ old(a[k]) = a[k]
6  {
7      a[i] := value;
8  }
```

Figure 3.7: An example Dafny program which swaps two elements into an array

3.3.3 Implicit Semantic Constraints

Although Dafny's implementation and specification do not contain undefined behaviour, there are a number of implicit semantic constraints which are only checked for by the verifier and not the compiler. Since a Dafny program is only valid if it verifies successfully, it is acceptable for the compiler to assume these constraints are met. Therefore, compiling programs which do not meet these constraints does not generate any warnings or errors, but may lead to undefined behaviour or runtime exceptions in the target backend – for example, `x / 0` would be invalid since it would trigger a divide-by-zero exception or arithmetic overflow, depending on the backend. It is therefore important for programs generated by *fuzz-d* to avoid these behaviours, since it will not be using the verifier when testing the compiler. A complete list of such behaviours is detailed in Table 3.1.

Type	Implicit Semantic Constraints
Arithmetic	Division or modulo by zero Arithmetic underflow/overflow Negative sequence/array initialiser length
Memory Access	Out-of-bounds array/sequence access Accessing an uninitialised identifier Invalid Datatype Destructor
Control Flow	Infinite loops Unreachable code Invalid for-loop range

Table 3.1: Summary of implicit semantic constraints by type in Dafny

3.4 Existing Testing Mechanisms

Previous work on testing the Dafny verifier and compiler led to the creation of XDsmith [3], which uses the fuzzing library Xsmith [35] to randomly generate annotated Dafny programs, testing the soundness and precision of the Dafny verifier and the correctness of the Dafny compiler. Due to the limitations of Xsmith, this is done using a subset of the Dafny language (*XDafny*) which includes simple language features such as basic types and data structures (Booleans, integers, strings, arrays, sets etc.), methods and functions, but omits more complex features such as loops, recursion, inductive datatypes and object-oriented features.

XDsmith tests the Dafny verifier using heuristics to generate programs with a known verification outcome, and the compiler is tested using differential testing. A generated program is compiled into each of the supported target languages and the outputs of each of these are compared to check for differences (which may possibly represent a bug).

In implementing *fuzz-d*, we aim to take inspiration from XDsmith since it lays good groundwork for testing the Dafny workflow and was able to identify a range of different bugs across the verifier and compiler. However, XDsmith is not without limitations and in particular we will consider and aim to avoid the following:

- Due to the limitations of the underlying framework, XDsmith has left a large portion of the Dafny language features untested, which the implementation of *fuzz-d* will be focused towards, in particular object-oriented structures, heap mutation and loops.
- Dafny's different compiler implementations for its 6 target languages all share a large proportion of the codebase for the compilation workflow (Figure 3.1). As such, the differential testing performed by XDsmith is unlikely to be able to identify bugs in stages prior to the final, language-specific AST transformations. For example, if a bug were contained in the SPC stage, all backends would include this behaviour and therefore the outputs would not differ during differential testing. *fuzz-d* will use differential testing over Dafny's backends, but will also aim to identify mechanisms which overcome its associated limitations, allowing *fuzz-d* to test stages throughout the compilation workflow.

4 | Design of *fuzz-d*

4.1 Overview

fuzz-d is a software tool intended for use in automatically testing the Dafny compilation workflow. It is implemented in Kotlin due to its powerful blend of functional and object-oriented features, and its concurrency APIs which prove useful in optimising the testing process.

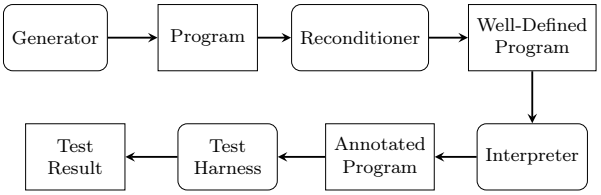
The design of *fuzz-d* consists of six core components, each of which contribute towards two core fuzzing workflows: one for testing the Dafny compiler and its backends (Figure 4.1), and another for performing testing over the Dafny verifier (Figure 4.2).

The components can be summarised as follows:

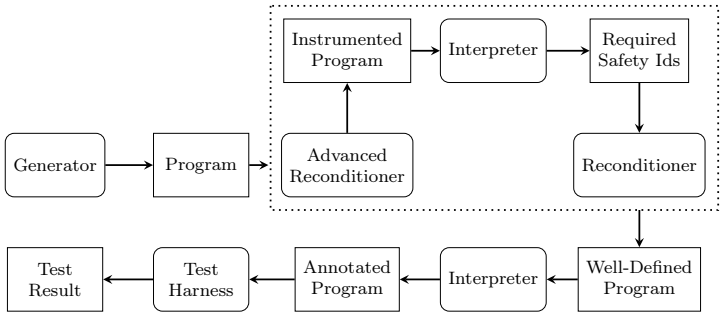
- A **program generator** for randomly producing Dafny programs to be tested.
- A **reconditioner** for transforming generated programs into a form which meets the implicit semantic constraints of the Dafny compiler. This includes an **advanced reconditoner** which can optionally be invoked to reduce the reconditioning transformations to only those which are strictly necessary.
- A **test harness** which concurrently verifies, compiles and/or executes generated Dafny programs, collecting and checking their outputs.
- An **interpreter** that simulates execution of a Dafny program to identify the expected output for a program, used by the test harness as a reference oracle for the Dafny backends and for filling in specification constructs with values in verifier testing.
- A **mutator** used to select and mutate specification constructs in an annotated program, taking a verified program and transforming it into one which should fail verification.
- A Dafny **parser** which converts Dafny programs into the internal AST representation used by *fuzz-d*. It can be used for reconditioning and interpreting arbritrary Dafny programs, which is particularly useful during test case reduction.

In both workflows, the generator produces test cases which are then reconditioned. The reconditioned program is passed to the interpreter to evaluate its expected output and also annotate the program with print statements (compiler testing) or correct, verifiable specification constructs (verifier testing). When testing the compiler, the test harness will differentially test the program across the interpreter output and Dafny backends. When testing the verifier, it must first check the original program verifies, since it is common for the Dafny verifier to be unable to verify some programs without additional help. If it does verify, the mutator will be invoked to create a number of invalidated programs which are equivalent except from one specification construct which has been mutated such that it is no longer correct. The test harness is then reinvoked to ensure that these programs do not verify.

This chapter will describe the design decisions involved in creating each of the above components, and how these affect their role in the overall workflows of *fuzz-d*.



(a) *fuzz-d* – workflow for fuzzing the Dafny compiler backends with standard reconditioning



(b) *fuzz-d* – workflow for fuzzing the Dafny compiler backends with advanced reconditioning

Figure 4.1: *fuzz-d* – workflows for fuzzing the Dafny compiler backends

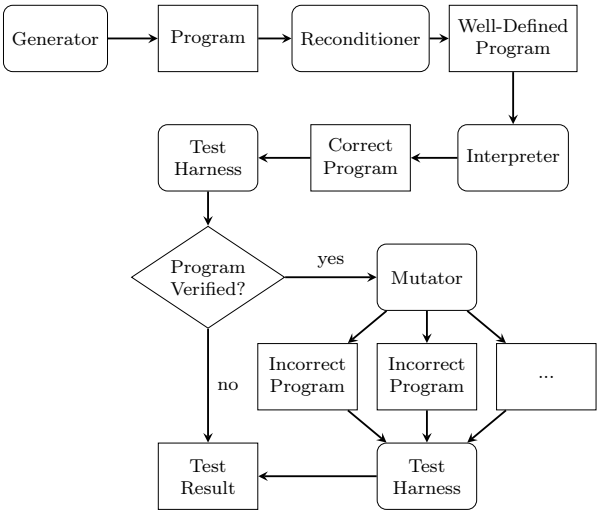


Figure 4.2: *fuzz-d* – workflow for testing the Dafny verifier

4.2 Generator

4.2.1 Generation Approach

fuzz-d takes a grammar-aided generative approach to compiler testing. Its generator is capable of randomly generating programs which are well-formed and well-typed, but may not be able to meet Dafny's implicit semantic constraints.

The design of the *fuzz-d* generator considers Dafny to be composed of four main types of language feature: top level components (such as definitions for classes and datatypes, functions and methods), types, expressions and statements. The generator selects from these language features to build an abstract syntax tree (AST) in a top-down fashion, which can then be output to a file by "pretty-printing" each AST element into its corresponding Dafny syntax, or further transformed by other processes within *fuzz-d*.

Generation starts at the top level, creating a `main` function, then generating its body and so on. It is separated into two core functions: one for generating statements and another for

generating expressions. These use a context-aware selection manager which identifies available *productions*, based on the current generation state, and performs a random weighted selection from these, calling into the generation function for the corresponding selected production. Since *fuzz-d* takes an on-demand approach to generation (Section 4.2.4), it is common for generation to be paused at a particular point so required elements (such as class definitions, datatypes, functions, identifiers etc.) can be generated on-demand when no suitable ones are available.

4.2.2 Generation Context

While building an AST, it is important to store the current generation state, including available variables, functions, classes etc., so that the generator can easily access any needed information. *fuzz-d* stores generation state in a *generation context*, which includes three symbol tables storing variables, functions/methods and top-level structures respectively, as well as information about program state which is useful for selection, such as statement/expression depth. The generation context is immutable, changed only via its internal functions (e.g. `increaseStatementDepth()`) which return a new instance of the context with updated values, and is passed to all generation-related functions in the core generator.

Symbol Table

fuzz-d uses a symbol table to store information about available variables, represented by a tree-based structure where each node stores a table of variable information. As with compiler design, symbol tables naturally fit the problem of random program generation since the tree structure encapsulates the idea of program scopes. In Dafny, like most high-level languages, variables initialised inside a scope do not exist in outer scopes, and therefore it is important for the generator to have separation between scopes, as different variables will be available in different scopes. To this extent, on entering generation in a new program scope (e.g. in an if/while statement), *fuzz-d* will create a new symbol table with the current symbol table as the parent node – once the scope is exited, the child symbol table can be discarded as any information which would survive outside the inner scope is contained in the parent symbol table.

Internally, the symbol table stores variables in a map from types to a list of variables of that type. This is the opposite to compilers which typically map from variable name to type, and is done to easily support the generator queries. These are based on the required type, rather than variable name: the generator will query for a particular type and then select randomly from the list of variables which are returned. There is a special case for instances of classes, traits and datatypes, for which the symbol table also stores related properties, such as available class/datatype fields. During queries, these are combined with their respective instance to form a list of compound variables (e.g. `c.f`) which is then searched over for the target type.

Function Symbol Table

Unlike variables, function (and method) definitions are not dependent on program scopes. However, since *fuzz-d* supports generation of object-oriented features, there are contexts where different functions may be visible, making functions context-dependent. *fuzz-d* stores functions in a *function symbol table*, which is also in a tree-based structure: the root node stores functions and methods at the top level of a program and each child node stores functions available inside a particular class context. This separation is required since inside a class context, a class's functions are available to call by name, whereas outside a class context they are only callable via a class instance (e.g. `c.m()`).

Global Symbol Table

fuzz-d stores language structures which are not dependent on program contexts in a global symbol table – this stores lists of available class, trait and inductive datatype definitions. Since these definitions always exist in the top level of a program – they are *globally available* – there is no need for the global symbol table to have a tree-based structure.

Program State Information

As well as information surrounding available structures, it is also necessary for the generation context to store information about the current point the generation is at – this information is used by the selection manager to make decisions about which productions are available in the current context. This information includes:

- **Statement and Expression Depth** – Generated programs are constrained in size by preventing statements and expressions from becoming too large. Therefore, if the current statement/expression depth has reached the maximum permitted depth, then productions cannot be chosen if they would increase this (e.g. binary expressions or if statements).
- **Class Generation Depth** – There exists an edge case within generation where class generation, due to being on-demand, can become recursive and potentially infinite. This happens when we are generating a method body within a class context, and decide to generate a new class definition (where the same can happen again). By tracking the depth of class generation, we can gradually reduce the probability of generating new class definitions, thus constraining program size and avoiding potentially infinite generation.
- **Generation Flags** – There are some language features where generating within their context requires disabling other language features to ensure validity. For example, Dafny's parallel assignments require that the right-hand-side is not allowed to be effectful. The generation context uses these flags to track when such language features are being generated for, so that the selection manager can avoid selecting any productions which would invalidate the program based on this context.

4.2.3 Context-Aware Selection Manager

Throughout the generation process, decisions need to be made in order to determine how the generation should continue – we can consider the generation process as a *decision tree*, where each point in the tree represents a selection from a number of generation options. Following the semantics of Dafny, in some program contexts it would not be valid to make certain decisions – for example, placing an effectful expression as the left-hand-side of a binary expression. Therefore, these decisions need to be made in a *context-aware* fashion to ensure that they always result in a valid program. Being context-aware also allows decisions to be made in such a way that enforces the constraint of program sizes through maximum expression and statement depths.

fuzz-d implements the decision-making process through a context-aware selection manager, which controls all generation-based decisions from selecting types, expression and statement productions to choosing the size of a method body, the number of parameters in a function signature or simply the value to place inside an integer literal. When a list of options is available, for example selecting the type of expression to generate, the selection manager performs a *random weighted selection*, where some of the options are more likely than others following their weightings. While selections could be made uniformly with each option having equal probability, in practice this was found to lead to very large programs being generated that could not be compiled within a fixed time period – it was very likely to generate statements and expressions that increase the depth, therefore often reaching the maximum constraints. This is problematic since

decreasing the maximum depth in response makes programs less interesting. Using probabilistic selection therefore allowed a *decay* to be introduced so that expressions and statements which increase the depth could become gradually more unlikely as depth increased, such that program interestingness is maintained while the maximum depth is rarely reached.

Probability Managers

Selection weightings are controlled internally by probability managers. This separates the selection manager from the probabilities and makes the weightings easily configurable. Different probability managers are implemented by *fuzz-d* to support a range of "modes":

- A `BaseProbabilityManager` is used by the standard generation approach, using a range of production weightings which have been adjusted throughout testing and identified as producing expressive, idiomatic programs.
- A `RandomProbabilityManager` which randomly assigns each production a random probability on instantiation, fixed for the duration of program generation. This is used by the swarm testing flag (`-sw`). It aims for some language features to become significantly less likely, therefore testing a smaller subset of the supported language in more detail. It can also be provided with a list of productions to disable to further focus testing onto a smaller subset of features.
- A `VerifierProbabilityManager` used for generating programs intended for testing the verifier. This is provided with one of the two above types as a base, and overrides some of their values to disable certain language features, such as classes and comprehensions, which the verifier isn't powerful enough to verify without additional lemmas and predicates.

4.2.4 On-Demand Generation

During generation, it is likely that a random program generator will come to a point where the information it requires is not available – for example, if it tries to generate an `int` type identifier when none have been declared so far. At this point, the generator can either fail and stop generation, or pause and generate the missing production on-demand, inserting it somewhere before the production currently being generated. Within *fuzz-d*, we take the latter approach as it enables more flexible generation and greater language expressiveness. To do so, it makes a number of accommodations within its generator.

The simplest on-demand case is when a top-level language structure (e.g. a class, function, inductive datatype etc.) is required but not available. In this case, the generator will pause in the current context, generate the required top-level structure and add this to the global symbol table before resuming generation of the current production. The top-level structure is automatically added to the AST at the end of the generation, since they are transferred directly from the global symbol table onto the AST.

To handle missing variables, the generator first generates a declaration to create a variable of the required type and adds this to the symbol table. The generator then aims to insert this declaration directly above the current production in the program, and does so by considering each generation not as generating a single AST element, but as generating an element of the required type along with its *dependencies* (any productions the generator had to create on-demand for this production). Functions generating an expression-type production therefore return a pair containing the expression itself, and a list of its dependencies (which are always statements). Similarly, functions generating statement-type productions return a list of statements, where the last is the required statement production. Taking this approach allows the AST to easily be built including any on-demand dependencies by concatenating the statement lists together.

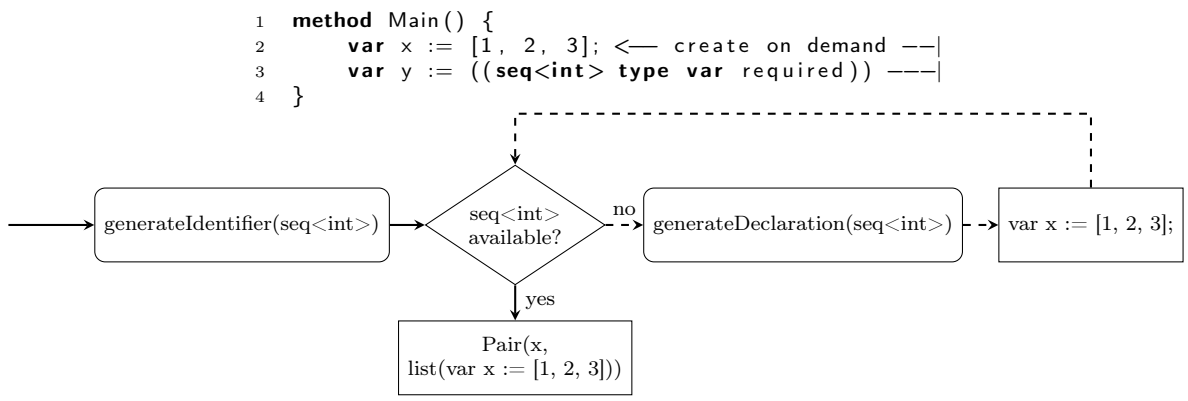


Figure 4.3: An example of creating an identifier on-demand in *fuzz-d*.

Figure 4.3 shows an example of how an identifier might be created on demand with *fuzz-d*. It tries to generate an identifier of type `seq<int>` on line 3, but such an identifier hasn't been created yet. Therefore, it generates the declaration `var x := [1, 2, 3];` which makes available an identifier of `seq<int>` type. The function `generateIdentifier` can then select `x` for the production and returns it alongside a list containing the declaration of `x` as its dependencies.

4.2.5 Handling Top-Level Structures

Generating Functions and Methods

Generation of functions and methods is supported in *fuzz-d* with the aim of testing the mechanisms behind function calls, such as evaluating and passing parameter references/values. During generation, functions and methods are handled using their signature – they are seen as expressions of a certain type, which require a number of expressions as parameters in order to obtain the resulting value. With the exception of within traits and classes, they are generated entirely on-demand when the generator selects a function call production. This is in order to minimise test case size, avoiding generating functions which may ultimately not be called.

To generate a function/method, the generator will check if one is already available given the current context – if not, it creates a signature on-demand. The bodies of functions and methods are populated at the end of the generation process since this allows for increased program expressiveness, where they have access to all other top-level structures.

```
1  method m0(p0 : int, p1 : int, globalState: GlobalState) {
2      ...
3      var v27 := m1(globalState);
4      ...
5      var v34 := m2({v27}, globalState);
6      ...
7  }
8
9  method m1(globalState: GlobalState) returns (r0: int) {...}
10
11 method m2(p0 : set<int>, globalState: GlobalState) returns
       (r0: bool) {...}
```

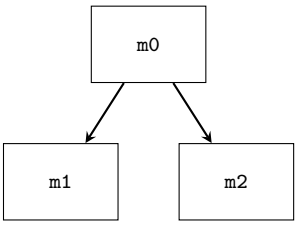


Figure 4.4: An example showing how *fuzz-d* forms call graphs to avoid recursion and mutual recursion in generated programs.

In order to ensure its generated programs will terminate, *fuzz-d* avoids generating functions bodies that introduce recursion or mutual recursion between functions. When generating function bodies, therefore, the generator creates a *dependency graph* modelling the call structure

between methods/functions, where a function is *dependent* on another when it calls that function. Functions are only available to be called from within a function body when that call does not introduce a cycle into the call graph. Figure 4.4 demonstrates how a call graph would be built for a generated program segment. Here, the bodies for `m1` and `m2` would not be able to call to `m0` since this would introduce a cycle.

Generating Traits and Classes

Classes and traits were targeted as a feature to implement in *fuzz-d* due to their complexity, particularly in that Dafny has to ensure class representations are consistent across backends. Since they are stored on the heap, inconsistencies are possible in how they are referenced, accessed and stored.

fuzz-d models classes and traits following their possible members, which can be any number of fields, functions or methods – *fuzz-d* stores separate lists for each of these. The only difference between the internal representations of classes and traits is that classes need to provide bodies for functions and methods, whereas traits only store their signatures. Both can also optionally *extend* a number of traits, which for classes requires values to be passed for their fields on instantiation.

Generation of classes and traits takes place in an on-demand fashion when a class instantiation or object-type production is selected. The following process is repeated to generate the required extended traits, functions, methods and fields:

1. Select the number of members to generate
2. Call into the corresponding generation function to create each member
3. If generating a class, populate the bodies of any member functions and methods while class context information is available

Alongside class definitions, *fuzz-d* implicitly generates a default constructor which requires parameter values to be passed for all class fields and implemented trait fields on instantiation.

```
1  trait T0 {
2    var f21 : int
3    function fm3(globalState: GlobalState): int
4  }
5
6  trait T1 extends T0 {
7    var f19 : int
8    method m0(p0: int, globalState: GlobalState) returns (r0: multiset<bool>)
9  }
10
11 class C10 extends T1 {
12   var f28 : bool
13   constructor (f28 : bool, f21 : int, f19 : int) {
14     this.f28 := f28;
15     this.f21 := f21;
16     this.f19 := f19;
17   }
18
19   function fm3(globalState: GlobalState): int {...}
20   method m0(p0: int, globalState: GlobalState) returns (r0: multiset<bool>) {...}
21   method m8(p0: bool, p1: int, globalState: GlobalState) returns (r0: array<bool>) {...}
22 }
```

Figure 4.5: An example of two traits generated by *fuzz-d*, and a class which extends them (function contents omitted due to size).

Figure 4.5 shows an abbreviated example from a program generated by *fuzz-d*, where class `C10` extends trait `T1`, and therefore also extends `T0`. It provides definitions for their members, as

well as its own members (e.g. `m8` and `f28`) which don't belong to the traits.

Generating Inductive Datatypes

Like classes and traits, inductive datatypes are a complex data structure which are non-trivial to implement within a compiler, therefore it is possible that bugs may exist in their implementation, particularly across Dafny's backends. *fuzz-d* generates inductive datatypes in order to investigate their correctness, since they are a core feature of Dafny's rich type system. There are a lot of features revolving around their use cases, yet they are noted by the developers as having weaknesses in their implementation [32].

fuzz-d generates datatypes on-demand when the generator selects a datatype instantiation production and none are available. It does so by producing a number of constructors, each of which can have a number of generated fields (including 0). Since the power of inductive datatype definitions lies in their ability to be recursive (for example inductive definitions of lists and trees), *fuzz-d* implements a heuristic to allow for datatypes to be recursive. This is handled by creating a datatype AST element with non-recursive constructors, and then optionally creating a recursive constructor with a single field and appending it to the list of constructors. It is necessary to do this after generating the other constructors to ensure that there is a *base case* – a non-recursive constructor – and that the generator has access to the datatype construct for creating the field. Figure 4.6 shows an example recursive inductive datatype created by *fuzz-d*, where constructor `DC17` is the recursive constructor created by the heuristic.

```
1 datatype D6 = DC16(cf34: int, cf35: bool, cf36: T0) | DC15(cf33: C1) | DC17(cf37: D6)
```

Figure 4.6: An example inductive datatype created by *fuzz-d*

Simulating Global State

Whereas some high-level languages support mutable global variables, in Dafny global variables can only be constant. However, within fuzzing, having mutable global variables can be very useful as they can be updated in all scopes, making it more likely that data from inner scopes will reach the checksum (Section 4.3.2) at the end of the program. This data might otherwise have been lost on exiting the progam scope if it was only assigned to variables local to that scope. Consequently, mutable global variables can make it more obvious to identify miscompilation bugs which occur in inner scopes.

Taking inspiration from CLsmith's global struct concept [36], *fuzz-d* achieves a similar notion to mutable global variables by utilising its support for classes to generate a *global state* class which stores a number of fields of different types. The class definition is created at the start of generation, and is initialised with a number of values at the start of the main function. This instance is passed around the program as a parameter to function and method calls, such that it is always available to be read from/written to. Figure 4.7 shows an example of a global state class definition and how this is instantiated at the start of the main function.

4.2.6 Other Generation Techniques

While *fuzz-d* handles generating top-level structures in a more unique way compared to other random program generators, its handling of generating statements and expressions is very similar. Rather than go into depth discussing generation techniques for these language structures which are already well-documented, the following sections will focus only on more unique aspects of generation specific to how *fuzz-d* must handle expressions and statements.

```
1  class GlobalState {
2    var f0 : array<bool>
3    var f1 : seq<bool>
4    constructor (f0 : array<bool>, f1 : seq<bool>) {
5        this.f0 := f0;
6        this.f1 := f1;
7      }
8  }
9
10 method Main() {
11     var v0: array<bool> := new bool[12];
12     var globalState := new GlobalState(v0, [false, false]);
13 }
```

Figure 4.7: An example global state class definition/instantiation generated by *fuzz-d*

Handling Types

Being able to generate and select types randomly is important within *fuzz-d* since this forms the basis of its expression generation: a *target type* is chosen randomly within a given context and available expression-type productions are identified and consequently selected from using their respective weightings.

fuzz-d supports the majority of Dafny's core types, including ints, booleans and chars, as well as more complex data structure types such as arrays, sequences, maps, sets and multisets. Data structure types are selected recursively by the selection manager, which allows for multi-level datatypes, for example `map<map<int, bool>, seq<int>>`. Like selection of expressions/statements, these are limited by tracking type depth which reduces the selection probability of a data structure type using a *decay* function. Due to differences in implementation across the backends, particularly related to printing, *fuzz-d* omits real types and constrains character generation to alphabetic values.

Type Inference in Declarations

In Dafny, declarations typically take the form `var <identifier>: <type> := <expr>;`. However, Dafny allows for the type annotation to be omitted, instead using its built-in type inference to detect the type of the right-hand-side expression. Since omitting type annotations has proven successful in detecting type inference bugs in popular JVM languages [37], *fuzz-d* omits providing declarations with the expression type in order to test Dafny's type inference capabilities. It will only annotate a declaration with the type if it is known that Dafny will be unable to infer it due to limitations in its implementation – this is particularly common for nested collection types.

Inserting Assert Statements

Assert statements are required for testing the verifier, in order to ensure that the verifier can check that certain logical statements hold throughout the program. To solve the problem of determining where they should be placed in the program, they are treated like other statements, inserted probabilistically during program generation when their production type is selected. *fuzz-d* aims to generate assertion statements of the form `assert(<expr> == <value>)`. It is easy to generate an arbitrary expression for the left-hand-side; however, since the generator does not track variable values (this is done by the interpreter), it cannot fill in the right-hand-side with a value for which it knows the assertion would hold. Instead, it therefore places a temporary right-hand-side which is the same as the left-hand-side to form `assert(<expr> == <expr>)`, and the right-hand-side is replaced by the interpreter (Section 4.3.3).

Ensuring Termination of Loops

fuzz-d supports both for-loops and while loops in ways which are guaranteed to terminate. This restriction is necessary since test cases which do not terminate cannot currently produce meaningful results for *fuzz-d* – they would become meaningful if *fuzz-d* supported a heuristic which could determine whether programs should terminate. Dafny's for-loops require integer bounds which ensures termination; however, there is no such guarantee for while loops with arbitrary conditions. Therefore, to ensure termination of while loops, *fuzz-d* implements a loop counter heuristic.

Each while loop generated by *fuzz-d* has an arbitrary boolean condition for maximum expressiveness, but alongside the loop, a counter is defined and incremented on each loop iteration. If this loop counter exceeds some defined maximum value – i.e. the maximum allowed number of iterations is reached – then the program exits the loop via a `break` statement.

```
1  var i5 := 0;
2  while (f22)
3      decreases 5 − i5
4  {
5      if (i5 ≥ 5) {
6          break;
7      }
8
9      i5 := i5 + 1;
10     // loop body
11 }
```

Figure 4.8: An example counter-limited while loop generated by *fuzz-d* (body omitted).

Figure 4.8 shows an example of how a terminating while loop would be generated by *fuzz-d* – in this example we also supply a `decreases` clause which is provided for the verifier to prove termination of the loop, alongside a small maximum number of iterations to enable the interpreter to provide invariant annotations for each iteration of the loop (Section 4.3.3).

Pattern Matching

An important use case related to inductive datatypes is the ability to pattern match over them and detect which constructor they are using. *fuzz-d* implements support for pattern matching through match statements, which allow for a control flow path to be taken based upon a datatype instance, and match expressions which select a value based upon the instance instead. *fuzz-d* generates for these by identifying all possible constructors for the given datatype instance, providing for each of these a sequence of statements (or expression) to be taken if the instance matches that constructor. If the datatype contains fields, these are made available by the pattern matching for use in the following scopes – this is demonstrated in Figure 4.9 on lines 3 and 4.

```
1 match v17 {
2     case DC0(cf0, cf1, cf2, cf3) ⇒
3         var v18: map<int, bool> := map[cf3 := cf2];
4         cf1 := fm0(−|v2|, 1) − cf1;
5         r0 := ¬v1;
6     case DC1() ⇒
7         r3 := v2[safeArrayIndex(v0, |v2|)];
8 }
```

Figure 4.9: An example of a match statement generated by *fuzz-d*.

4.2.7 Supported Language Features for Verification

Although the *fuzz-d* generator supports a number of language features, some currently cannot be used in the verifier testing workflow. This is because Dafny is not powerful enough to be able to perform verification over them without further assistance, for example needing additional predicates, proofs or lemmas to verify such features. The complete list of features supported by the generator but not the verifier testing workflow is shown in Table 4.1. *fuzz-d* aimed to minimise this list by first generating verified programs with all supported generation features, gradually removing those which could not be verified. While it may be possible to implement proof constructs to assist Dafny in verifying these language features, it was not deemed within the scope of this project, whose primary focus is testing the Dafny compiler.

Language Feature
Object-Oriented Features
Collection Comprehensions
Collection Binary Operators (e.g. subset, prefix)

Table 4.1: Language features currently supported for generation which cannot be verified without additional help

4.3 Interpreter

Differential testing over the Dafny backends has a significant limitation that it can only detect miscompilation bugs in the final, target-language-specific transformations over the AST, since the rest of the codebase is shared between backends (Section 3.1). To overcome this, *fuzz-d* implements an interpreter which independently evaluates the expected output of generated Dafny programs. This acts as a reference oracle for the output of the backends with the aim that even if all backends display the same behaviour, if the interpreter achieves a different output then this could signify either a bug in the interpreter, or a miscompilation bug deeper in the Dafny compiler.

While the interpreter is used primarily to evaluate the expected output of a program (Section 4.3.1), its knowledge of runtime values of variables is useful for annotating programs with meaningful print statements (Section 4.3.2) and filling in the template assert statements and specification constructs produced by the generator (Section 4.3.3).

4.3.1 Evaluating Expected Output

The interpreter is implemented as a traversal over the given AST, utilising symbol tables to store a map from variables to their values as it passes through the program's methods and scopes. Where variables are uninitialised, for example a method's named return values are uninitialised on entry, their value is stored as `null`.

Interpreting programs in *fuzz-d* follows common interpreter design, interpreting a program line-by-line (starting with the `main` function), visiting the loop body on each iteration and method bodies on each method call. However, *fuzz-d* treats functions and method calls on a class instance (e.g. `c.m()`) slightly differently – it is necessary to carry around a class context, which is used to make field values for that particular class instance (e.g. `c`) available to the interpreter. Expression evaluation also follows common design, although Boolean expressions are evaluated using short-circuit semantics, and division and modulus use Euclidean semantics following their approach in Dafny.

4.3.2 Print Checksum

In order for a program to be able to be tested using differential testing, it needs to produce a meaningful output. *fuzz-d* aims to produce meaningful output by printing the values of all variables available in the top-level of the main function at the end of the program. As much as possible, it aims to do this by printing the variable, i.e. `print x` for some variable `x`. However, for array and collection types, printing in Dafny is not consistent across backends and therefore this is not possible. To be able to check the values of these types, *fuzz-d* uses information known about their values to generate a meaningful print statement. This therefore requires the interpreter since the generator does not have this information available.

Print statements are generated by the interpreter after it reaches the end of the program, and they are returned as a separate list alongside the expected output of the program. Although most print statements are able to meet the form `print x`, those for array and collection types require handling differently: for array types, it identifies the array indexes which have been initialised and prints these individual indexes, and for collection types, it finds the expected contents of the collection and uses a *display expression* to create a new instance of the collection with these contents, printing whether or not this is equal to the original.

4.3.3 Filling in Specification Constructs

Providing valid specification constructs is key to being able to create a program which Dafny is able to verify. *fuzz-d* uses the generator to insert template loop invariants, pre- and post-conditions and assertions. However, these templates are trivial, simplifying to `true`, and may not provide the Dafny verifier with enough information to be able to verify a program – for example, a method post-condition which simplifies to `true` provides no information about the method's return values, therefore Dafny cannot reason about the values of any variables which take the return values of a method call, since they could be taking any value. Consequently, the interpreter implements heuristics to allow for specification constructs to be filled in with known variable values, such that they should provide enough information for Dafny to be able to verify resulting programs.

Most variables and expressions in a program generated by *fuzz-d* have a single known value, and this allows most instances of template assertions (of form `assert(<expr> == <expr>)`) to be completed by replacing the right-hand-side with the interpreted value of the left-hand-side – Figure 4.10 shows two assertions whose left-hand-sides have a single known value, and thus their right-hand-sides have been replaced with their evaluations.

```
1 method Main() {
2     var v0 := 386;
3     assert ((v0 + (v0 - 0x27c)) = 136);
4     var v1: multiset<seq<string>> := multiset{["w", "a"]};
5     assert ((v0 * (v0 - |v1|)) = 148610);
6 }
```

Figure 4.10: An example showing completed assertions for expressions with a single known value

However, in some program contexts, variables and consequently expressions can take multiple values. In particular, variables within method bodies can take different values depending on the parameters the method is called with, and similarly variables within loops can take different values depending on how many times the loop body has been executed if they are evaluated using members of the loop's modset. Therefore, the interpreter must ensure that any specification constructs within these contexts – including invariants, pre- and post-conditions and assertions – are completed in a way which is dependent on the context.

To fill in specification constructs in these contexts, *fuzz-d* uses conjunctions of implies clauses:

the left-hand-side of the implies clause places conditions on the variable values, and the right-hand-side provides a condition expected to be true given these values. An implies clause is created for each possible combination of variable values, with the expectation that one of them will be met while those that aren't can still be verified. Figure 4.11 shows how implies clauses are used to annotate specification constructs in method contexts, where different parameter values can lead to different conditions or condition outcomes.

```
1  method m0(p0: bool, p1: int) returns (r0: seq<bool>, r1: bool)
2    ensures (p0 = true ∧ p1 = 16 ⟹ r0 = [true, false] ∧ r1 = false)
3    ensures (p0 = false ∧ p1 = 16 ⟹ r0 = [false, false] ∧ r1 = false)
4  {
5    var v0: set<int> := if (p0) then {p1} else {0x2ac, p1, 546, p1 + 43};
6    assert (p0 = true ∧ p1 = 16 ⟹ (|v0| < 3) = true) ∧
7            (p0 = false ∧ p1 = 16 ⟹ (|v0| < 3) = false);
8    r0 := [p0, false];
9    r1 := p1 > (p1 + p1);
10 }
```

Figure 4.11: An example of using implies clauses to verify in method contexts with multiple known variable values.

Equally, Figure 4.12 demonstrates the use of implies clauses to construct loop invariants – note that here, we only need a single invariant using this mechanism since the loop is never entered.

```
1  var i0 := 0;
2  while(false)
3      decreases 5 − i0
4      invariant (0 ≤ i0) ∧ (i0 ≤ 5)
5      invariant (i0 = 0) ⟹ v14 = true ∧ v13 = 709 ∧ v4 = [false]
6  { ... }
```

Figure 4.12: An example of using implies clauses to verify in while loop invariants.

4.4 Reconditioner

Since the generator produces arbitrary programs which may not meet the implicit semantic constraints of the Dafny compiler (Section 3.3.3), *fuzz-d* implements a reconditioner module whose responsibility is to take generated programs and transform them such that they meet these constraints. Transformations are applied to the following language features:

- **Integer Division and Integer Modulo** – in order to avoid division by zero
- **Array and Sequence Indexing** – in order to avoid out of bounds accesses
- **Sequence Initialiser Length** – ensuring the length provided to a sequence initialiser is always positive

Avoiding these behaviours is important since programs exhibiting them can produce runtime exceptions or inconsistent results across the Dafny backends.

Reconditioning is performed by *fuzz-d* in two ways: *standard reconditioning* (Section 4.4.1), and *advanced reconditioning* (Section 4.4.2). The integration of both of these into the *fuzz-d* workflow is shown in Figure 4.1.

4.4.1 Standard Reconditioning

Standard reconditioning transforms generated programs into a valid form by traversing the AST recursively and inserting inline calls to *safe wrapper functions* for operations which can introduce behaviour that would lead to runtime exceptions or inconsistent outputs. The aim

of these functions is to take the parameters for these operations and analyse if performing the operation would result in invalid program behaviour – if this is the case, a default or "safe" value is returned, otherwise the operation is performed and the result returned. Figure 4.13 shows the functions generated alongside each program which are used for these standard reconditioning calls, and Figure 4.14 shows how these are applied to recondition a program segment. Here, inline calls are made to `safeIndex` to ensure that the indexes into array `v52` are valid.

By default, standard reconditioning applies the safe wrapper functions at all instances where program behaviour may be invalid; however, the reconditioner is configurable and allows a list of ids to be passed in so as not to generate the safe wrapper calls for these ids. This is particularly useful for the final stage of advanced reconditioning – more detail is provided below.

```
1  function abs(x: int): int {
2    if (x < 0) then −1 * x else x
3  }
4
5  function safeIndex(x: int, length: int): int
6    requires length > 0
7  {
8    if (x < 0) then 0 else if (x ≥ length) then x % length else x
9  }
10
11 function safeDivisionInt(x1: int, x2: int): int {
12   if (x2 = 0) then x1 else x1 / x2
13 }
14
15 function safeModuloInt(x1: int, x2: int): int {
16   if (x2 = 0) then x1 else x1 % x2
17 }
```

Figure 4.13: Safe wrapper functions used by *fuzz-d* for standard reconditioned programs

```
1  if (v48.f14 in v51) {
2      var v52: array<bool> := new bool[29];
3      v52[19] := v47 ⟹ fm3(v46, 0x389, globalState);
4      v45 := v52[19];
5  }
```

(a) Program segment before reconditioning

```
1  if (v48.f14 in v51) {
2      var v52: array<bool> := new bool[29];
3      v52[safeIndex(19, v52.Length)] := v47 ⟹ fm3(v46, 0x389, globalState);
4      v45 := v52[safeIndex(19, v52.Length)];
5  }
```

(b) Program segment after reconditioning

Figure 4.14: An example showing how reconditioning transforms a program segment

4.4.2 Advanced Reconditioning

Since standard reconditioning applies safe wrapper calls at all instances of possibly invalid program behaviour, this can limit the expressiveness of the resulting program – it may unknowingly obscure possible bugs by inserting a safe wrapper call in a place where the original behaviour was valid and triggered a bug. For example, in Figure 4.14, the safe wrappers are called regardless of the fact that the initial program is valid without them. Therefore, we take inspiration from Even-Mendoza et al. [20] in implementing an optional advanced reconditioning workflow as an extension to standard reconditioning. This performs dynamic analysis on generated programs, aiming to only place safe wrapper calls where they are necessary, i.e. where if they were omitted, the resulting program would be invalid.

```
1  method requireSafetyId(id: string, advancedState: AdvancedReconditionState) {
2    if (¬(id in advancedState.state) ∨ ¬advancedState.state[id]) {
3      print id, "\n";
4      advancedState.state := advancedState.state[id := true];
5    }
6  }
7
8  method advancedAbsolute(p1: int, advancedState: AdvancedReconditionState, id: string)
       returns (r1: int) {
9    if (p1 < 0) {
10     printSafetyId(id, advancedState);
11     r1 := −1 * p1;
12   } else {
13     r1 := p1;
14   }
15 }
16
17 method advancedSafeIndex(p1: int, p2: int, advancedState: AdvancedReconditionState, id:
       string) returns (r1: int) {
18   var b1, b2 := p1 < 0, p1 ≥ p2;
19   if (b1 ∨ b2) {
20     printSafetyId(id, advancedState);
21     r1 := if (b1) then 0 else p1 % p2;
22   } else {
23     r1 := p1;
24   }
25 }
26
27 method advancedSafeModInt(p1: int, p2: int, advancedState: AdvancedReconditionState, id:
       string) returns (r1: int) {
28   if (p2 = 0) {
29     printSafetyId(id, advancedState);
30     r1 := p1;
31   } else {
32     r1 := p1 % p2;
33   }
34 }
35
36 method advancedSafeDivInt(p1: int, p2: int, advancedState: AdvancedReconditionState, id:
       string) returns (r1: int) {
37   if (p2 = 0) {
38     printSafetyId(id, advancedState);
39     r1 := p1;
40   } else {
41     r1 := p1 / p2;
42   }
43 }
```

Figure 4.15: Advanced safe wrapper methods used by *fuzz-d* for advance-reconditioned programs

Advanced reconditioning is performed by *fuzz-d* in three stages. First, it transforms the generated program into a valid, intermediate AST which is instrumented with print statements, enabling detection of exactly which safe wrapper calls are needed. It then uses the runtime output of this intermediate representation to identify the wrappers which are needed, and finally performs standard reconditioning to insert inline wrapper calls at the places where these occur in the original AST.

Intermediate Transformation

In performing the intermediate transformation, *fuzz-d* aims to annotate the generated program with calls to *advanced safe wrapper* functions (Figure 4.15) which replicate the behaviour of standard reconditioning's safe wrappers, except that in the case that the wrapper is needed – i.e. the inputs would lead to invalid behaviour – then they should print a *safety id* which is unique to that particular application of the advanced safety wrapper. If a safety id is printed, therefore, this signifies that the wrapper application it refers to is required for the program to

be valid.

Since the same wrapper applications can be reached multiple times, for example if they are applied within loops, *fuzz-d* implements a heuristic to minimise the program output by only printing safety ids which have not already been printed – it is only necessary for a safety id to be printed once in order for that wrapper application to be required. This is achieved by passing around an *advanced recondition state* which stores a map from safety ids to boolean flags signifiying the id is required (which is true iff the safety id has been printed). The advanced safe wrappers update this state each time an id is needed, by calling `requireSafetyId` in Figure 4.15 which handles both printing the id and updating the state. In order to facilitate this, the state is passed around the intermediate program in a class instance: the definition of the advanced recondition state class is shown in Figure 4.16, and is necessary in order to pass around the map and perform updates on it *by reference*.

```
1  class AdvancedReconditionState {
2    var state : map<string, bool>
3    constructor (state : map<string, bool>) {
4      this.state := state;
5    }
6  }
```

Figure 4.16: Class definition used in advanced reconditioned programs to track printed safety ids

Compared to the transformations applied by standard reconditioning, those required for the intermediate transformation are a lot more intrusive to the original program. This is because Dafny considers print statements to be *effectful*, therefore the advanced safe wrappers must be represented by methods, not functions as with standard reconditioning. Consequently, the wrappers can no longer be called inline within expressions, since Dafny does not allow calls to methods within expressions. Instead, temporary variables must be created to store the results of advanced safe wrapper calls, and these temporary variables are then inserted inline where the wrapper functions would have otherwise been placed. Figure 4.17 shows an example of using temporary variables to transform two index operations.

```
1  v24[188] := v0[|v25|];
```

(a) Original statement

```
1  var t156 := advancedSafeIndex(188, v24.Length,
       advancedState, "safety110");
2  var t157 := advancedSafeIndex(|v25|, |v0|,
       advancedState, "safety111");
3  v24[t156] := v0[t157];
```

(b) The same statement in the intermediate program

Figure 4.17: An example showing how advanced reconditioning transforms a program using temporary variables for safe wrapper calls

Furthermore, since functions can potentially have calls to the safe wrappers, all functions in the original program must be converted to a method of equivalent behaviour, so that they can call the advanced safe wrappers. This is relatively straightforward and involves creating a method with the same parameters and return type, assigning the function expression to the named method return value. Any necessary advanced safe wrappers can then be inserted above the return value assignment. Function call results are again assigned to temporary variables, which are inserted inline where the function calls are in the original program.

By far the most intrusive transformation, however, is for comprehensions and set/array initialisers. In order to replicate these in a way which supports calls to safe wrapper methods, they must be converted into a while loop which builds an equivalent structure, where requiring use of a safe wrapper in building the equivalent structure implies the original comprehension/initialiser needs the safe wrapper call.

Figure 4.18 shows how a map comprehension with potentially unsafe behaviour is transformed into a while loop via advanced reconditioning. Since the original comprehension is over elements in data structure `v58`, the loop must also iterate over these elements – it does so by creating a clone of `v58` (`t84`), which can be mutated in the update on line 10 to remove the visited element without affecting the original data structure. The map starts as empty and is populated with elements via the update on line 9.

```
1  var v64 := map[v2 := map v63 : int | v63 in v58 :: (v63 / v47[959]) := (v2)];
```

(a) Before advanced reconditioning

```
1  var t83 := map[];
2  var t84 := v58;
3  while (|t84| ≠ 0)
4      decreases |t84|
5  {
6      var v63 :| v63 in t84
7      var t81 := advancedSafeIndex(959, v47.Length, advancedState, "safety61");
8      var t82 := advancedSafeDivInt(v63, v47[t81], advancedState, "safety62");
9      t83 := t83[t82 := v2];
10     t84 := t84 - {v63};
11 }
12 var v64 := map[v2 := t83];
```

(b) After advanced reconditioning

Figure 4.18: An example showing how comprehensions need to be transformed during advanced reconditioning.

Collecting Intermediate Program Output

When the intermediate program is run, any *live* code in the program which requires a safe wrapper call will result in corresponding ids being printed. These ids need to be collected and parsed by *fuzz-d* in order to identify the wrappers which need to be inserted in the final stage of the advanced reconditioning process.

The output of the instrumented intermediate program can be collected in two ways: using the built-in interpreter (Section 4.3), or compiling and executing the program over one of Dafny's backends. *fuzz-d* opts to use its in-built interpreter to obtain the output for two reasons:

1. While executing a compiled Dafny program takes around the same amount of time as *fuzz-d* would to interpret it, the compilation of the program itself is very time-consuming, often taking around 15-30 seconds for an average generated program. By using the interpreter, *fuzz-d* can avoid this time penalty, significantly reducing the time taken per test case.

2. Using the interpreter instead of one of the Dafny backends increases the scope for being able to find bugs. For example, a latent bug might exist in one of Dafny's earlier AST transformations before the backends. This could cause the wrong control flow branch to be taken at one point in the program, and since the bug is in an earlier transformation, all backends would therefore exhibit the same behaviour. If the interpreter was used instead of one of the backends to collect the safety ids, it would hopefully take the correct branch instead. Therefore, the branch taken by the backends would be *dead*, thus no safety ids for the branch would be output and it would not be annotated with safe wrappers. When the backends would run the final advance-reconditioned program, they would take the incorrect branch and any invalid code would cause easily identifiable runtime exceptions.

Minimising Safe Wrappers with Safety Ids

Once the safety ids have been collected from the output of the intermediate program, they are passed to the standard reconditioner, along with the original generated AST. The reconditioner will, for each possible instance of invalid behaviour, check if its associated safety id is contained in the given list – if it is, it will generate the required safe wrapper call, otherwise leaving the AST element unchanged. The output of this recondition pass is the final, *advance-reconditioned* AST.

4.5 Mutator

Within the verifier testing workflow, if a program is able to be successfully verified by the Dafny verifier, it is then passed to the *mutator*. This takes the verified program and recursively traverses over the AST, randomly selecting one annotation to make invalid, with the aim of ensuring that the Dafny verifier can identify the invalid assertion. Our motivation for invalidating programs is that it is much more impactful to identify programs which Dafny incorrectly verifies, rather than programs which Dafny is unable to verify – these could just be due to limitations in the verifier.

Annotations are made invalid either by flipping the equality sign of the condition, from `<expr> == <value>` to `<expr> != <value>`, or instead replacing the condition with `false`. Once an annotation has been invalidated, the rest of the AST is returned in the original, unmutated form, and the mutated program is cross-checked against the verifier using the test harness to check that Dafny can successfully detect the invalid assertion.

```
1 if (v1) {
2     v0 := if (true) then v0 else -1;
3     var v10 := multiset{-v0};
4     assert (v10 = multiset{-1983});
5 }
```

(a) Before mutation

```
1 if (v1) {
2     v0 := if (true) then v0 else -1;
3     var v10 := multiset{v0};
4     assert (v10 ≠ multiset{1983});
5 }
```

(b) After mutation

Figure 4.19: An example showing how the mutator invalidates assertions

Figure 4.19 shows how the mutator can apply invalidations to generated programs – here, the assertion on line 4 is inverted such that `==` becomes `!=`.

4.6 Test Harness

The test harness in *fuzz-d* is responsible for taking a reconditioned program and either checking that it verifies, or compiling and executing it in order to run differential testing. It utilises concurrency as much as possible, running compilation/verification/execution commands in parallel to reduce the time taken to test each program.

The compilation/execution workflow is shown in Figure 4.20, which splits into threads where for each thread, the program is first compiled to the corresponding target language, and if the compilation is successful, the output compiled program is executed. For both of these stages, the outputs of standard input and standard error are collected and these, along with the exit codes for each stage, form the test result for the backend. Once this process has been completed for all backends and the threads are joined, the results are analysed to identify any crashes during compilation or execution, and differential testing is run over all backends for which the execution terminated successfully. The overall results are output to a log file, `fuzz-d.log` which contains the individual test results for each backend and overall conclusions from the analysis and differential testing.

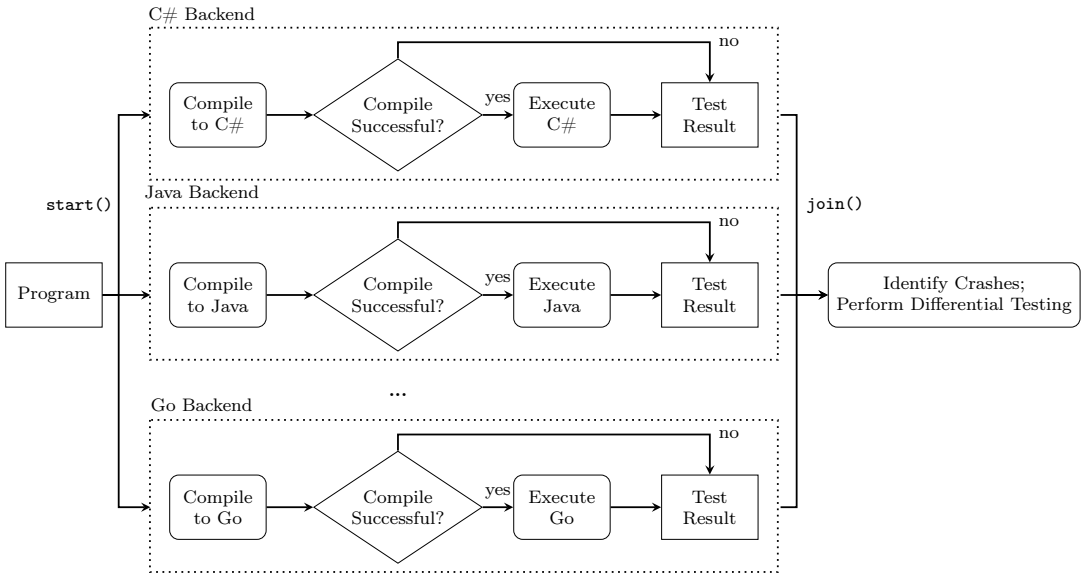


Figure 4.20: *fuzz-d* test harness workflow for testing the Dafny compiler backends

A similar approach is used when verifying the mutated programs generated in the verifier-testing workflow, shown in Figure 4.21. It takes the list of mutated programs, on successful verification of the original program, and passes each of these to a thread which runs the Dafny verifier on it, collecting the standard input/error and exit code to form a verification result. After the threads are joined, it analyses these verification results for each program to identify if any successfully verified when they should have failed from the mutation. Like the compiler workflow, the results are output to `fuzz-d.log`, including the outputs for each mutated program and whether any incorrectly verified.

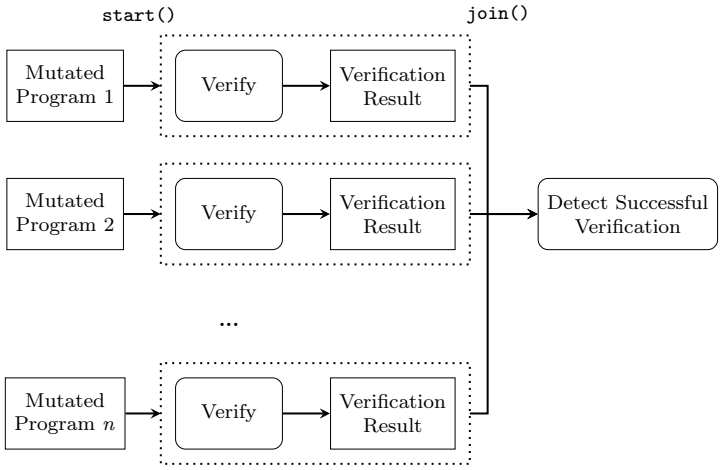


Figure 4.21: *fuzz-d* test harness workflow for testing the Dafny verifier with mutated programs

4.7 Parser

Since *fuzz-d* takes a language-agnostic approach to test case reduction, it is possible that during the reduction process, invalid program behaviour is introduced. To avoid a reduced program containing invalid program behaviour, therefore, it is important to recondition programs as they are reduced to ensure that reduced programs are always valid. In order to support reconditioning during the reduction process, *fuzz-d* therefore implements a parser, overloading a generated

ANTLR visitor. This allows it to parse arbitrary Dafny programs (using the same language subset) into its internal AST representation, which can then be transformed via reconditioning. This also allows programs to be re-interpreted during reduction in cases where the interpreter output is important to the interestingness test result.

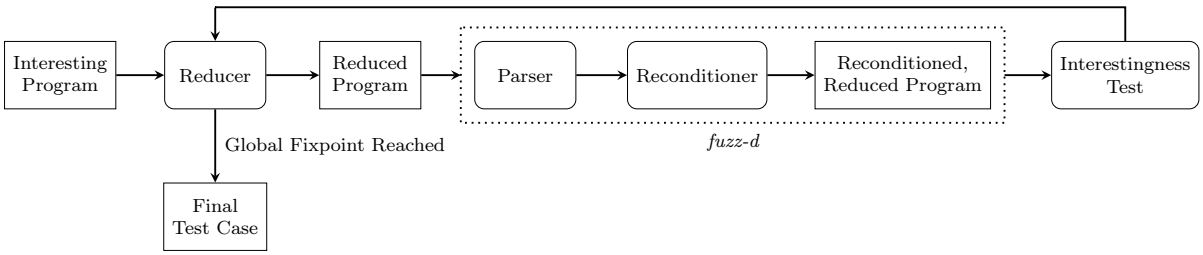


Figure 4.22: Workflow for reducing a program, maintaining validity through reconditioning

Figure 4.22 shows how the *fuzz-d* parser can be used for reconditioning a program within a reducer workflow. Note that reconditioning is performed on each reduced program, rather than reconditioning a reduced program, since this makes it much easier to ensure program validity and avoids the reducer interfering with reconditioning constructs.

Additionally, the ANTLR grammar used to generate the parser visitor is also used to integrate with PERSES [28], allowing for language-agnostic program reduction. With this, we can reduce Dafny programs using more language-specific transformations based on the ANTLR grammar, optimising reduction and making it much more likely to reach a test case's minimal form, compared to more general, language-inspecific transformations applied by other language-agnostic reducers such as C-Reduce [27].

5 Evaluation

The success of the project will be evaluated following the research questions detailed below. Where relevant, we will draw comparison to existing testing mechanisms for Dafny, such as XDsmith and Dafny's regression test suite, to identify both potential areas of improvement in *fuzz-d* and areas of weakness in the existing mechanisms.

RQ1 – *Is fuzz-d effective in identifying bugs in Dafny's compilation workflow?*

RQ2 – *What level of coverage of the core Dafny compilation source code is fuzz-d able to achieve?*

RQ3 – *What level of mutation coverage can fuzz-d achieve over the source code for the core compilers?*

RQ4 – *How effective is advanced reconditioning in assisting fuzz-d's compiler testing?*

RQ5 – *Is the verifier testing workflow a valuable addition to fuzz-d?*

5.1 RQ1: Evaluating *fuzz-d*'s Ability to Identify Bugs

We performed consistent testing with *fuzz-d* in a largely uncontrolled manner throughout development, over multiple versions of Dafny. As of the current version of *fuzz-d*, each test case takes approximately 20-40 seconds, depending on its size and included language features.

From its testing, *fuzz-d* has identified a range of bugs across components of Dafny's compilation workflow. These include both compile-time crashes and miscompilations, where an executable was produced by the compiler but it exhibited unexpected runtime behaviour (either crashing or producing an incorrect output).

Component	Crash	Miscompilation	Total
Parser	1	0	1
Resolver	3	0	3
Verifier	2	1	3
Compiler	1	27	28
Total	7	28	35
Total (Deduplicated)	5	13	18

Table 5.1: Summary of bugs identified by *fuzz-d* across Dafny's core components

Backend	Miscompilation
C#	5
Java	12
Python	4
Go	3
JavaScript	3
Total	27

Table 5.2: Distribution of Compiler Miscompilations across Backends

Table 5.1 summarises the bugs found by *fuzz-d* over Dafny's components, and Table 5.2 further expands on how the compiler miscompilations were distributed over the backends. Overall, *fuzz-d* identified 35 instances of unexpected behaviour relating to a particular language feature. However, a large proportion of these individual unexpected behaviours were related, either from the same issue existing across multiple backends, or different errors in a single backend relating to the same implementation detail. Therefore, it is necessary to perform deduplication, shown in Table 5.1, after which *fuzz-d* found 18 unique, unrelated bugs. Of these, 4 were identified as already reported, thus a total of 14 reports [38] were submitted to Dafny developers.

These reports are summarised in Table 5.3. Of the bugs reported, one was labelled a duplicate of another bug (originally identified by XDsmith) and the parser crash was labelled as an error-reporting issue instead of a bug. All others are either confirmed or awaiting confirmation, with

Issue	Status	Type	Component	Description
#4061	Unconfirmed	Invalid Code	Compiler (Java)	Generated code with invalid use of variables
#4011	Unconfirmed	Wrong Code (Soundness)	Compiler (C#)	Incorrect equality of multisets with 0-cardinality elements
#4007	Fixed	Invalid Code	Compiler (Python)	Invalid syntax in generated code
#3988	Confirmed	Invalid Code (Soundness)	Compiler (C#)	Runtime cardinality limit for multisets
#3988	Fixed	Invalid Code (Soundness)	Compiler (Python)	Runtime cardinality limit for multisets
#3987	Unconfirmed	Invalid Code	Compiler (Python, JS)	Referencing undeclared variable in generated code
#3969	Confirmed	Crash	Verifier	Assertion failure during translation
#3950	Confirmed	Incorrect rejection	Resolver	Incomplete type checking for multiset operations
#3932	Won't Fix	Error Reporting	Parser	Handling of a clash in the grammar
#3910	Unconfirmed	Invalid Code	Compiler (Java, Go, C#)	Multiple issues using variables in pattern matches
#3906	Unconfirmed	Crash	Verifier	Boogie – Internal translation error
#3874	Fixed	Invalid Code (Soundness)	Compiler (Python)	Multiset equality issues
#3873	Confirmed	Invalid Code (Soundness)	Compiler (JS)	Type representation issues for maps with array keys
#3856	Duplicate	Wrong Code (Soundness)	Compiler (JS, Go)	Incorrect internal representation of maps
#3854	Confirmed	Invalid Code	Compiler (Java)	Multiple issues related to type representation and class casting.

Table 5.3: Summary of bugs reported to Dafny developers

three Python backend miscompilations having been fixed. The following sections will highlight some examples of these bugs.

All Backends: Forall Expression Inside Match Statement

Of the language features tested by *fuzz-d*, pattern-matching-related features were among those which caused the most issues. The case where a forall parallel assignment was inserted inside a match statement was particularly problematic, causing issues across all 5 backends: in 3 backends (C#, Java and Go), Dafny produced invalid code that caused exceptions during compilation by the relevant backend, while for the interpreted backends (Python and JavaScript), this resulted in runtime exceptions. This was the only identified miscompilation bug which affected all backends, and since the bug affects all backends, it is likely it is contained in an earlier compiler stage.

```
1  datatype D = A | B
2
3  method Main() {
4      match A {
5          case A ⇒
6              var a: array<int> := new int[24](i1 ⇒ i1);
7              forall i2 | 0 ≤ i2 < a.Length {
8                  a[i2] := i2;
9              }
10         case _ ⇒ {}
11     }
12 }
```

Figure 5.1: Test case inserting a parallel assignment inside a match statement. Taken from [39]

Previous bugs found pointed to backends not being able to correctly handle variable use inside match expressions and statements [40], and this particular combination of features (shown in Figure 5.1) emphasises this. Dafny will translate the above program correctly until line 7, where for `a.Length`, it uses an invalid reference for `a` in what seems to be an error in its naming mechanism. This results in all backends producing a form of undeclared variable exception, for example in JavaScript this causes the exception `ReferenceError: _2_a is not defined` (the JavaScript generated code is available in Appendix A.1 for correlation).

This bug was detected during an 8 hour testing campaign on the most recent version of Dafny (at the time of testing), and it currently remains possible to replicate on the newest commit of Dafny at the time of writing. *fuzz-d* was able to identify it as a result of its support for pattern matching over datatypes, and parallel assignment over array indexes. The reduced test case was able to be created manually since the error clearly signposts where the issue occurs, thus it took less time to understand the issue and write a reduced case than it would have done to run the reducer.

C# and Python Backends: Runtime Cardinality Limit of Multisets

An interesting edge case in the C# and Python implementations of multisets was identified by *fuzz-d*. When trying to take the modulus of a multiset whose size is greater than the maximum supported integer value, runtime exceptions are thrown due to an arithmetic overflow from trying to fit the modulus into an integer type. The other three backends are able to handle this case using Dafny's `BigInteger` implementation, which allows Dafny to have (theoretically) unbounded integer values, therefore this is clearly a missed implementation detail in the C# and Python backends.

```
1  method Main() {
2      var x: multiset<int> := multiset{};
3      x := x[1 := 18446744073709551616];
4      print |x|;
5  }
```

Figure 5.2: Test case for demonstrating runtime cardinality limit of Python multisets. Taken from [41]

Figure 5.2 shows the test case which caused this error for Python – it sets multiplicity of the value `1` in multiset `x` as 2^{64}, which is greater than the maximum allowed value for an index-type integer. Consequently, when the test case was run, the error `OverflowError: cannot fit 'int' into an index-sized integer` was thrown.

Like the case above, this error was identified during a testing campaign and consequently reduced using PERSES. Once the reduced test case made it clear how the error was happening, some manual reduction was performed to further simplify the test case – previously, a while loop was responsible for creating such a large multiset and not a single update. The GitHub issue [41] was originally tracking both backends, but Dafny developers split the issue so as to fix the Python implementation [42], while the C# implementation is confirmed and awaiting a fix.

Java Backend: Incompatible Types

As seen in Table 5.2, the Java backend was by far the most affected by miscompilation errors. A lot of these were related crashes during compilation of the `jar` executable, and could be contributed to weaknesses in the type and class definitions surrounding the Java code generation implementation. In particular, conversions between internal type representations would fail and result in a crash often labelled as `error: incompatible types`. Of the 12 Java miscompilations identified, three result in the above error, and another test case was identified as being related (4 total).

```
1  method Main() {
2      var x := {map[0 := 1]}
3          + {map[1 := 1]};
4  }
```

(a)

```
1  method Main() {
2      var x := new int[10];
3      var y := {{x}};
4  }
```

(b)

```
1  method Main() {
2      var x := new int[22];
3      var y := if (true) then
4          multiset{x} else
5          multiset{x};
6  }
```

(c)

Figure 5.3: Three test cases which crash with a unique incompatible types error when compiled to Java. Taken from [43]

Figure 5.3 shows three test cases causing the three instances of the incompatible types error. Each of these involves operations with nested collections, and crashes with an error message involving wildcard generics corresponding to the nested collection type. They were detected during an 8 hour campaign and reduced entirely with PERSES, and were able to be detected

thanks to *fuzz-d* supporting a multitude of collection types and their operations. The bugs have been confirmed on the GitHub issue [43] but have yet to be fixed.

Boogie Internal Translation Error

Alongside several miscompilations across the Dafny backends, *fuzz-d* was also able to identify crashes in the resolver, verifier and compiler. Figure 5.4 shows an example of a test case which identified a miscompilation bug in the verifier, triggering an internal translation error related to Boogie. The program is able to be translated into Boogie (`.bpl`) files, but these fail Boogie type checking at the comparison between the two sets (`{1} != {1}`) and consequently the internal translation error is thrown. Although it was possible that this issue was related to Boogie, investigating different possible similar test cases and related Dafny source code suggested that the issue is more likely to be related to the Dafny verifier.

```
1 method Main() {
2     while (true ∧ ({1} ≠ {1})) { }
3 }
```

Figure 5.4: Test case causing an internal Boogie translation error. Taken from [44]

This test case was the simplest identified among the bugs reported, and highlights the effectiveness of the reduction process to produce a reduced test case in its smallest possible form.

Summary

The above examples taken from the selection of reported bugs clearly demonstrate, in response to **RQ1**, that *fuzz-d* is effective in identifying bugs in both the Dafny compiler and verifier, capable of generating a diverse range of programs which combine language features in interesting ways to capture multiple new edge cases in the Dafny language. Naturally, there will remain latent bugs within Dafny which *fuzz-d* has so far been unable to detect, either from not implementing those language features, or from its random decision causing generation to move away from these bugs. However, given the number of bugs *fuzz-d* has been able to detect in such a short time frame using only a small subset of the language's features, there is promising evidence to suggest that for testing over a longer time period, with a greater subset of the Dafny language, *fuzz-d* would be able to further increase its ability for generated programs to find bugs.

5.2 RQ2: Evaluating Coverage Achieved by *fuzz-d*

A key part of evaluating the ability of *fuzz-d* to test Dafny lies in understanding how much (and which parts) of the Dafny source code *fuzz-d* is able to test. In order to do so, we analysed the coverage which test cases generated by *fuzz-d* were able to achieve over the Dafny source code, in particular the core module (`DafnyCore`) which contains the majority of the source code for both the compiler and verifier – other Dafny modules are not relevant to this project.

5.2.1 Experimental Setup

Being able to measure code coverage requires having access to an instrumented version of the Dafny compiler source, and this was possible using the coverlet tool [45] which instruments Dafny's .NET binaries to calculate line and branch coverage over a particular run, optionally combining results over multiple runs. Coverlet-generated reports of the `cobertura` format can be converted into html using the `reportgenerator` tool to provide easily viewable results.

To evaluate coverage of test cases generated by *fuzz-d*, we performed a controlled experiment, running testing campaigns using *fuzz-d* over a coverage-instrumented version of the Dafny com-

piler. These ran for a fixed 12 hour period against the latest Dafny commit at the time of testing, and were repeated two additional times to handle (to some extent) the random nature of compiler testing. The campaigns were run on Linux virtual machines with x86_64 architecture, 4 cores, 8GB RAM and running Ubuntu 20.04.

Alongside *fuzz-d*, we ran the same experiment using XDsmith, the results of which act as a baseline for fuzz-testing Dafny, and we also calculated the coverage of the Dafny integration test suite for comparison, highlighting possible areas of improvement for *fuzz-d*, and possible areas of weakness in the Dafny test suite. Since Section 5.5 focuses on evaluating verifier testing, these experiments focus only on compiler testing, i.e. they do not involve the Dafny verifier. For the Dafny integration test suite, this therefore involved filtering the test cases to avoid those which do not use the compiler.

5.2.2 Results

Table 5.4 shows the line and branch coverage measured using the experiments detailed above. Note that branch coverage is calculated with respect to line coverage, thus it is not necessarily comparable across tools. We compare and contrast these results below, comparing first *fuzz-d* and XDsmith, then *fuzz-d* and the Dafny integration test suite.

Run	Line (%)	Branch (%)
1	32.56	30.58
2	32.58	30.58
3	32.56	30.55
Avg	32.57	30.57

(a) Coverage results for XDSmith

Run	Line (%)	Branch (%)
1	46.40	40.56
2	46.46	40.40
3	46.30	40.06
Avg	46.39	40.34

(b) Coverage results for *fuzz-d*

Line (%)	Branch (%)
75.46	40.74

(c) Coverage Results for Dafny Integration Tests

Table 5.4: Coverage experiment results, showing line and branch coverage percentages

Evaluating *fuzz-d* against XDsmith

The results across the coverage experiments for *fuzz-d* and XDsmith show that, on average, *fuzz-d* was able to achieve 13.82% higher line coverage than XDsmith, marking an increase of 42.43%.

Inspection of the coverage report (summarised in Appendix B) revealed that *fuzz-d* is generally able to cover a larger and more diverse range of language features when compared to XDsmith. The largest differences between the two fuzzers stem from features which *fuzz-d* generates for while XDsmith does not. For example, *fuzz-d* is able to cover 510 of the 842 lines related to pattern matching (60.61%) on average, while XDsmith covers just 34 (4.04%) – Table 5.5 shows the difference between the two over these files.

File	Missing Lines	Coverage Difference (%)
DafnyCore/AST/Expressions/NestedMatchCase.cs	−7	+100.00
DafnyCore/AST/Expressions/NestedMatchCaseExpr.cs	−16	+94.11
DafnyCore/AST/Expressions/NestedMatchCaseStmt.cs	−28	+82.35
DafnyCore/AST/Expressions/NestedMatchExpr.cs	−48	+75.00
DafnyCore/AST/Expressions/NestedMatchStmt.cs	−69	+78.41
DafnyCore/CompileNestedMatch/MatchAst.cs	−86	+40.95
DafnyCore/CompileNestedMatch/MatchFlattener.cs	−263	+53.56

Table 5.5: Coverage comparison between *fuzz-d* and XDsmith for pattern matching constructs

The impact of increased language support is not only evident in the AST files for the features themselves, but its impact also accumulates into *fuzz-d* covering significantly more of the core compiler files than XDsmith. These files are responsible for interacting with, transforming and

File	Missing Lines	Coverage Difference (%)
DafnyCore/Compilers/Python/Compiler-python.cs	−287	+23.35
DafnyCore/Compilers/CSharp/Compiler-Csharp.cs	−618	+27.74
DafnyCore/Compilers/Java/Compiler-java.cs	−737	+23.41
DafnyCore/Compilers/GoLang/Compiler-go.cs	−639	+24.61
DafnyCore/Compilers/JavaScript/Compiler-js.cs	−388	+21.16
DafnyCore/Compilers/SinglePassCompiler.cs	−1014	+27.43

Table 5.6: Coverage comparison between *fuzz-d* and XDsmith over core compiler files

outputting AST elements for all types of language features, and in total *fuzz-d* was able to average 3683 more lines covered across these files, which represents around 4.65% of the entire DafnyCore module – Table 5.6 shows how this difference was distributed over the compiler files.

While *fuzz-d* has generally achieved higher coverage than XDsmith, there remain areas where it can improve, and where XDsmith was able to cover which *fuzz-d* could not. These areas of weakness are summarised by Table 5.7. Perhaps the largest area of weakness visible is that *fuzz-d* does not support *arrow types*, which are used by Dafny to represent both partial and total functions, while XDsmith does.

File	Missing Lines	Coverage Difference (%)
DafnyCore/AST/Types/ArrowType.cs	+25	−37.88
DafnyCore/Compilers/CSharp/CsharpBackend.cs	+7	−6.36
DafnyCore/Compilers/Java/JavaBackend.cs	+24	−18.75

Table 5.7: Coverage comparison between *fuzz-d* and XDsmith over files where *fuzz-d* was weaker

Interestingly, while *fuzz-d* achieves much higher coverage for the individual compiler files, it appears to have weaker coverage over the backend files, which are responsible for producing an executable for the target language. Further inspection showed this difference is related to errors while compiling the executable resulting in control flow paths being missed. For the Java backend, there is a missing path related to successful compilation, and since *fuzz-d* never took it this means the compilation of *every* `jar` file resulted in errors (possible bugs). On the contrary, for the C# backend there is a missing path corresponding to a compilation crash, meaning that all its programs successfully compiled into an executable for *fuzz-d* while at least one resulted in a crash or error for XDsmith. In the case of the Java backend, this result might suggest that there is a *fuzz blocker* (or multiple) which occurs so frequently that it prevents observing the runtime behaviour of the Java compiler – this could be resolved by identifying the fuzz blocker(s) and adjusting generation probabilities so as to avoid generating them when testing the Java backend.

Evaluating *fuzz-d* against the Dafny Integration Tests

Although *fuzz-d* was able to achieve coverage improvements over XDsmith, it is clear that there is still a lot more the tool could achieve, demonstrated by the Dafny integration test suite achieving 29.07% higher line coverage. A large portion of this difference follows from additional language features covered by the integration suite having impact across the codebase. The features which made the largest impact on the difference were related to Dafny's type system, including newtypes, bitvectors and co-inductive datatypes (which *fuzz-d* currently omits due to their limited functionality in executable code). It is also likely that some of the difference comes from the Dafny test suite having annotated programs, while *fuzz-d* does not annotate programs for compiler testing – although the programs were not verified, there remains portions of code in the compilers for handling annotations.

While it is interesting to identify missing language features as areas of improvement/future

directions for *fuzz-d*, in terms of evaluation it is much more useful to analyse the coverage differences over the features which were supported by both test mechanisms. Generally, *fuzz-d* is able to achieve a similar level of coverage for the language features that it does support – for example, the AST files related to for-loops, match statements/expressions and multiset displays all have the same coverage across both. There are, however, instances where the Dafny integration tests are able to have higher coverage for a particular feature, as a result of particular edge cases being accounted for in the tests but not *fuzz-d*. Consequently, areas have been highlighted where features are perhaps *under-implemented* in *fuzz-d*. An example of this is related to `forall` constructs, which have a variety of edge cases including allowing parallel assignment over class members for arrays of class instances, or calls to ghost methods (although the latter is verification only). Since *fuzz-d* does not implement for these edge cases, some blocks are consequently uncovered in related code (`ForallStmtRewriter`). A minimal program showing this uncovered edge case is demonstrated in Figure 5.5.

```
1  class C {
2      var f : int
3      constructor(f: int) { ... }
4  }
5  method Main() {
6      var c := new C(1);
7      var arr := new C[1][c];
8      forall i | 0 ≤ i < arr.Length {
9          arr[i].f := i + 1;
10     }
11 }
```

Figure 5.5: Minimal program demonstrating the omitted `forall` edge case.

On the other hand, a small number of instances have been identified where *fuzz-d* is able to cover code which the Dafny integration tests do not – these could signify weaknesses in the test suite. The full report showing these is available on Gist and summarised in Table 5.8. Overall, *fuzz-d* achieved greater coverage than the integration tests for two files, related to shallow equality of expressions, and 23 files were identified across DafnyCore where at least one line was covered by *fuzz-d* and not the integration tests. In the following paragraphs we analyse the root causes of some of the cases of lines missed.

DafnyCore/AST/ComprehensionExpr.cs: Cloning of Sequence/Multiset Bounded Pools
This file is responsible for representing comprehensions in the Dafny internal ASTs. They are considered by Dafny as being formed of three parts: a list of bound variables, a *range* which confines these variables to a finite range of values, and a *term* which represents the expression used to evaluate the comprehension's elements. The range can take multiple different forms, and each of these results in the possible range values being represented as a *bounded pool* – for example, providing an int range (e.g. `0 <= i < 10`) results in an `IntBoundedPool` while providing a data structure (e.g. `x in [1, 2, 3]`) results in a bounded pool corresponding to that data structure. The lines omitted by Dafny's integration suite result from an edge case related to the bounded pools for sequences and multisets.

Each bounded pool implements a function `Clone()` providing a deep copy of itself. This is necessary since the Dafny internal AST representation is mutable, meaning that if the current AST state needs to be cached or maintained for later use, it must be cloned so as not to be changed by later transformations. However, the Dafny integration tests omit testing the clone functions for comprehensions bounded by the contents of sequences or multisets. This is covered *fuzz-d* following its support of match statements and comprehensions, and a reduced test case obtained from *fuzz-d* covering cloning of sequence bounded pools is shown in Figure 5.6.

DafnyCore/Compilers/Python/Compiler-py.cs: Non-Sequentialisable Forall

File	Coverage Difference (%)	Lines Missed
DafnyCore/DafnyOptions.cs	+3.47%	4
DafnyCore/AST/Cloner.cs	+40.14%	1
DafnyCore/AST/Expressions/ComprehensionExpr.cs	+19.34%	6
DafnyCore/AST/Expressions/Expressions.cs	+20.63%	1
DafnyCore/AST/FreeVariablesUtil.cs	+8.27%	2
DafnyCore/AST/Grammar/Printer.cs	+34.54%	14
DafnyCore/AST/Statements/Statements.cs	+18.77%	3
DafnyCore/AST/Substituter.cs	+27.06%	11
DafnyCore/AST/Types/Types.cs	+23.88%	5
DafnyCore/Resolver/NameResolutionAndTypeInference.cs	+28.03%	6
DafnyCore/Resolver/Resolver.cs	+44.20%	4
DafnyCore/Resolver/TailRecursion.cs	+30.80%	6
DafnyCore/Verifier/Translator.cs	+37.18%	37
DafnyCore/Verifier/Translator.ExpressionTranslator.cs	+23.66%	13
DafnyCore/Parser.cs	+42.84%	1
DafnyCore/Scanner.cs	+23.53%	2
DafnyCore/Triggers/TriggerExtensions.cs	−3.69%	47
DafnyCore/Triggers/TriggersCollector.cs	−0.35%	12
DafnyCore/Compilers/DatatypeWrapperEraser.cs	+20.00%	1
DafnyCore/Compilers/Java/Compiler-java.cs	+32.08%	8
DafnyCore/Compilers/JavaScript/Compiler-js.cs	+40.02%	10
DafnyCore/Compilers/Python/Compiler-python.cs	+30.27%	3
DafnyCore/Compilers/SinglePassCompiler.cs	+29.40%	4

Table 5.8: Comparison between *fuzz-d* (baseline) and the Dafny integration test suite, showing the number of lines covered by *fuzz-d* that were missed by the test suite

```
method Main() {
    var a := [1, 2, 3];
    match true {
        case _ ⇒ var b := map x | x in a :: x := x * x;
    }
}
```

Figure 5.6: Smallest test case covering cloning of the `SeqBoundedPool` class

Although *fuzz-d* omitted some edge cases related to forall statements which the integration tests could cover, the inverse is also true: *fuzz-d* is able to cover non-sequentialisable forall statements within the Python compiler while the integration tests do not.

A non-sequentialisable forall statement is one where executing the assignments in a sequential order would result in a different outcome to executing the assignments in parallel. Figure 5.7 demonstrates a simple non-sequentialisable forall obtained from *fuzz-d*, where executing the assignments in parallel always takes `a[0]` as its initial value `2`, but executing them sequentially would update `a[0]` to value `4` on the first iteration, after which the results would be different.

```
method Main() {
    var a := new int[3][2, 3, 4];
    forall i | 0 ≤ i < a.Length {
        a[i] := i * a[0];
    }
}
```

Figure 5.7: Smallest test case covering non-sequentialisable forall statements in the Python compiler

DafnyCore/Compilers/SinglePassCompiler.cs: Missed Binary Operators
A number of the files identified to have missed lines were as a result of missing cases for binary operators, notably for explies (`<==`), iff (`<==>`) and map/multiset not equals (`!=`). The

`SinglePassCompiler` omits the last three of these in the function `CompileBinOp()`, and because of the significance of this function in the compilation flow, this means that that no programs testing the compiler in the integration test suite use these binary operators. Analysing backwards through the workflow, looking at the missing lines in the `Resolver` further reveals that the binary operators for explies and map not equals never even reach the resolver, thus no semantically valid programs in the integration tests contain these operators. Missing these operators could be argued as a potential weakness in Dafny, since binary operators are among the most commonly used language features and therefore it is important the tests are able to identify any compile issues related to all possible types. Figure 5.8 shows a simple test case where each line would introduce coverage for one of the above operators.

```
1 method Main() {
2     print true ⟸ false;
3     print true ⟺ true;
4     print map[1 := 1] ≠ map[1 := 2];
5     print multiset{1} ≠ multiset{2};
6 }
```

Figure 5.8: A small test case with statements to achieve coverage for missing binary operators

5.2.3 Summary

To answer **RQ2**, we have demonstrated in this section that *fuzz-d* is able to average 46.39% line coverage over the DafnyCore module when testing only the Dafny compilers, including over 50% line coverage in all core compiler files. As a result of using a highly flexible generational approach to enable the support of more complex language features, *fuzz-d* raises the bar over existing compiler testing mechanisms for Dafny; however, there naturally remains a large gap between *fuzz-d* and the overall coverage standard set by the Dafny integration test suite.

As with any compiler testing, it is inevitable that additional improvements can be made to *fuzz-d* by supporting more language features, in comparison to both XDsmith and the Dafny integration tests. By comparing *fuzz-d* to these over its supported features, however, we have also found more specific improvements that could be made, identifying edge cases which have resulted in certain features being under-implemented. Despite this, *fuzz-d* is still able to cover multiple areas of the DafnyCore module which the Dafny integration tests are unable to reach, highlighting potential weaknesses in Dafny's built-in testing mechanisms which we recommend are looked into by the Dafny developers.

5.3 RQ3: Evaluating Mutation Coverage Achieved by *fuzz-d*

In order to further understand the effectiveness of testing Dafny with *fuzz-d*, we performed *mutation testing* experiments over the Dafny compiler source code. Mutation testing is used to measure the quality of a given test suite by analysing its ability to detect small, functional changes in the system under test. Therefore, we formed a test suite of programs generated by *fuzz-d* and compared it to a similar process run over the Dafny tests and an XDsmith test suite.

5.3.1 Experimental Setup

We used a C# mutation testing tool, Stryker [46], to perform three fixed experiments evaluating mutation coverage using *fuzz-d*, XDsmith and the Dafny integration test suite respectively. Stryker identifies and injects mutations – including arithmetic, logical, initialisation and assignment-based – into the Dafny codebase, with the test suite then being invoked to see if it can *kill* the mutant (i.e. at least one test fails). Due to the compute-intensive nature of mutation testing – Stryker generates over 80,000 mutants over all of DafnyCore – it is necessary to limit

the scope of the experiment to mutations affecting only the core compiler files. This generates 13,390 mutations – any more would be computationally infeasible to test. Furthermore, we performed this experiment in a *best effort* approach as it was infeasible to repeat the runs in the limited time frame of this project (each run takes upwards of 7 days).

Generating Test Suites

Since compiler testing does not naturally involve working with test suites, it was necessary for us to use *fuzz-d* and XDsmith to form test suites representative of the capabilities of each tool. These consist of programs generated by the tool and their expected output over each backend. To make this as fair as possible in comparison with the Dafny integration test suite, we allowed the fuzzers to generate for the time taken to execute the Dafny tests, with the aim of all three test suites therefore having approximately the same amount of time to test Dafny.

5.3.2 Results

Table 5.9 shows the mutation results for each test suite, displaying the number of mutants which were either killed, survived or timed out, where the injected mutation caused the tests to run much slower than their initial, unmutated run.

Test Suite	**Killed**	**Survived**	**Timed Out**
fuzz-d	2335	0	11,055
XDsmith	13,335	0	55
Dafny Integration Tests	13,346	44	0

Table 5.9: Mutation coverage results for all three test suites

Test Suite	**Killed**	**Survived**	**Timed Out**
fuzz-d	0	0	2967
XDsmith	2925	0	42
Dafny Integration Tests	2923	44	0

Table 5.10: Mutation coverage results for all three test suites over `SinglePassCompiler`

Overall, we can see that the Dafny integration tests were able to kill almost all mutants, except for a handful which all lie in the `SinglePassCompiler` (Table 5.10). These survivors are located in functions relating to the compilation of function call expressions and direct comparisons of integer types with zero, which is handled as a special case within Dafny's `bool`-type expressions. Surviving mutations included string-based mutations which would invalidate the output program, and therefore it is likely that no Dafny compiler tests covering these functions target trying to create and run an executable, since these mutations would then be easily detectable. This is further backed by XDsmith being able to kill two of these string-based mutants, while the Dafny tests could not, as a result of the XDsmith test suite performing comparisons over the program output.

For both XDsmith and *fuzz-d*, we see that no mutants survived; however, the tests for some timed out – this is particularly an issue for *fuzz-d* where over 80% of all mutants timed out. This was caused by the timeout not being lenient enough, consequently we cannot get accurate coverage data for these mutants; however, given that XDsmith killed a similar number of mutants to Dafny, and that *fuzz-d* supports more language features, there is strong evidence to suggest that we could expect the results for *fuzz-d* to be at least as high as those for XDsmith.

Although it was infeasible to re-run the tests for *fuzz-d* in order to obtain more representative results over all compiler files, it was feasible for us to perform another mutation coverage run over

the `SinglePassCompiler` (2967 mutations). This is most interesting as it allows for comparison to the mutants which survived for the Dafny integration tests.

Test Suite	Killed	Survived	Timed Out
fuzz-d	2960	1	6

Table 5.11: Mutation coverage results for *fuzz-d* over `SinglePassCompiler` with an increased timeout

Re-running the mutation testing over `SinglePassCompiler` (Table 5.11) showed that *fuzz-d* is able to kill a much larger number of mutants than the Dafny integration tests. In particular, *fuzz-d* is able to kill all but one of the mutants which survived in Dafny's case – those which timed out were in unrelated parts of the codebase to the survivors. This further supports how it is likely the Dafny integration tests do not test the behaviour or output from functions relating to function calls and integer comparisons with zero – while reports from Section 5.2 show that these functions are covered during testing, mutation survivors suggest that they are not directly tested.

5.3.3 Summary

The above results show that, in response to **RQ3**, *fuzz-d* is effective in killing mutants, consisting of small, functional changes in the core Dafny compiler file, `SinglePassCompiler`. Despite obtaining inconclusive results for *fuzz-d* over the remaining compiler files due to experimental configuration issues, we have been able to show that *fuzz-d* can kill mutants which survive when tested with the Dafny integration test suite, thus identifying potential edge cases which have been missed by Dafny's built-in testing. As *fuzz-d* only supports a subset of Dafny's language features, there will remain some mutants which it cannot currently kill. However, there is already strong evidence (even from limited testing) to suggest that with increased language support, *fuzz-d* would become even more effective in identifying and killing mutants in the Dafny codebase.

5.4 RQ4: Evaluating the Effectiveness of Advanced Reconditioning

In evaluating the advanced reconditioning component of *fuzz-d* (Section 4.4.2), we aim to identify whether this novel development adds value to the testing performed by *fuzz-d*. We perform this evaluation in two stages:

(a) Surveying the extent to which advanced reconditioning increases the ability of *fuzz-d* to identify bugs.

(b) Measuring the coverage which *fuzz-d* with advanced reconditioning enabled is able to achieve over the DafnyCore source, comparing this to the standard tool.

5.4.1 Evaluating Bug-Finding with Advanced Reconditioning

Since the advanced reconditioning option builds upon the existing tested mechanisms for standard reconditioning, running *fuzz-d* with advanced reconditioning is therefore able to identify all bugs which are identified by the standard tool (detailed in Section 5.1). However, the advanced reconditioning mechanism was unable to identify any bugs which could not be found in programs generated with the standard tool. This is likely as a result of the Dafny compiler applying very few optimisations during AST transformations and code generation, unlike optimising compilers (such as the C compiler in the case of [20]) where removing the safe wrapper calls and replacing them with their corresponding expressions could trigger more optimisations.

Although running *fuzz-d* with advanced reconditioning enabled was unable to identify any further bugs, it made bugs which result in a wrong value much easier to detect and reduce than if the same bug only existed in a program generated by the standard tool. Since advanced reconditioning ensures that *live* code is annotated with safe wrappers (Section 4.4.2), when these wrong values resulted in a different control flow branch being taken (which should have been dead without the bug), invalid code along that branch would cause an exception. It is much easier to create an interestingness test to avoid bug slippage during reduction when there is a program crash versus a different output, therefore these bugs were much easier to reduce to a final test case.

5.4.2 Evaluating Coverage Achieved Using Advanced Reconditioning

Coverage using advanced reconditioning was measured using the same experiment detailed in Section 5.2. The results are shown in Table 5.12.

Run	**Line** (%)	**Branch** (%)
1	46.43	40.59
2	46.44	40.54
3	46.39	40.66
Avg	46.42	40.60

Table 5.12: Coverage results from running *fuzz-d* for 12 hours with advanced reconditioning enabled

The coverage results show that running *fuzz-d* with advanced reconditioning enabled achieved very similar results to the standard tool (Table 5.4). While the average branch coverage may suggest a slight improvement over those for *fuzz-d* in Table 5.4, analysis of the coverage reports does not indicate that the increase is as a direct result of the advanced reconditioning. The difference is more likely to result from the random decision making performed by *fuzz-d* covering more branches in these runs.

5.4.3 Summary

The above two sections demonstrate that, while running *fuzz-d* with advanced reconditioning does not decrease its effectiveness, it does not necessarily add value to it in terms of coverage achieved or new bugs identified. However, while the advanced reconditioning may not add measurable value to *fuzz-d*, it has been demonstrated to make identifying and reducing wrong value bugs easier when they are involved in the control flow of a program, and this could help make it more likely that future bugs in Dafny will be identified compared to using the standard tool. This, alongside the minimal impact that running advanced reconditioning has on the overall time taken for a test case (thanks to the interpreter), indicates that advanced reconditioning may therefore add value to *fuzz-d* after all. Moreover, *fuzz-d* currently supports only a limited subset of the Dafny language, thus it is possible that with additional supported language features, advanced reconditioning may be able to identify some bugs that the standard tool cannot.

5.5 RQ5: Evaluating the Verifier Testing Workflow

Similarly to how we evaluated advanced reconditioning, we use bug-finding and coverage metrics as a means to evaluate the effectiveness of *fuzz-d*'s verifier testing workflow.

5.5.1 Evaluating Bug-Finding with Verifier Testing

Bug finding with the verifier testing workflow was performed throughout development in the form of 8-hour testing campaigns. So far, no verifier bugs have yet been identified using this testing mechanism, and we attribute this to the combination of the following reasons:

Enhancement
Reasoning with binary operators on sets/multisets
Reasoning with the prefix operator for sequences
Reasoning about collection values defined within a comprehension

Table 5.13: Verifier enhancements identifiable using *fuzz-d*

- The verifier testing workflow was the most recent addition to *fuzz-d*, thus the total testing time with this workflow is less than one month. More time may be required for the random decision making to generate programs which trigger bugs in the verifier.
- Since *fuzz-d* primarily aims to test the compiler, it does not support an extensive set of Dafny's verification-oriented features, for example omitting object-oriented features such as classes and traits. Those it does support are arguably more common and therefore more likely to be well-tested, compared to generating programs using Dafny's more experimental verification features.

While the workflow has not yet identified any bugs, the development pattern used to create the supported Dafny subset has been useful in identifying potential *enhancements* for the Dafny verifier, represented by language features which the verifier is currently not powerful enough for. These are listed in Table 5.13 and surround operations related to collection-types – proving specifications using these requires additional axioms which Dafny does not currently implement.

5.5.2 Evaluating Coverage Achieved with Verifier Testing

To measure the coverage of the verifier testing workflow over DafnyCore, we will use the same experiment setup as in Section 5.2; however, since we are now testing a different part of Dafny, we must disable the compiler and all backends (using the `/compile:0` flag) and instead enable the verifier to run for each program.

Run	Line (%)	Branch (%)
1	24.46	21.32
2	24.52	21.35
3	24.54	21.37
Avg	24.51	21.35

(a) Coverage results for XDSmith

Run	Line (%)	Branch (%)
1	26.60	22.35
2	26.53	22.27
3	26.53	22.25
Avg	26.55	22.29

(b) Coverage results for *fuzz-d*

Line (%)	Branch (%)
58.83	37.85

(c) Coverage Results for Dafny Integration Tests

Table 5.14: Coverage experiment results, showing line and branch coverage percentages

Table 5.14 shows the coverage measured from verifier testing over XDSmith, *fuzz-d* and the verification-related Dafny integration tests. We will use the below sections to compare *fuzz-d* to both XDsmith and the Dafny integration tests.

Evaluating *fuzz-d* against XDsmith

The coverage results show that *fuzz-d* is able to achieve a marginal improvement over XDsmith in covering the DafnyCore module with verifier testing – an average line coverage improvement of 2.04%. Similarly to the compiler coverage results (Section 5.2), *fuzz-d* is able to make significant coverage improvements to XDsmith over code related to pattern matching features. However, this is largely located in the same AST files as previously, and it is more interesting to consider key code related to verification.

When analysing coverage over verifier files within the DafnyCore module (Table 5.15), it is evident that *fuzz-d* is able to achieve higher coverage over data types, statement translation and proof descriptors; however, it is slightly weaker in comparison for expression translation.

File	Coverage Difference (%)
DafnyCore/Verifier/Translator.DataTypes.cs	+17.25%
DafnyCore/Verifier/Translator.ExpressionWellformed.cs	−2.88%
DafnyCore/Verifier/Translator.TrStatement.cs	+14.67%
DafnyCore/Verifier/Translator.ExpressionTranslator.cs	−1.42%
DafnyCore/Verifier/ProofObligationDescription.cs	+5.84%

Table 5.15: Coverage comparison between *fuzz-d* and XDsmith over core compiler files

Compared to XDsmith, *fuzz-d* supports verification over match statements and while statements (including break clauses) and this led to higher coverage in both `TrStatement` and `ProofObligationDescription`. The latter file handles proof descriptions for ensuring program validity and termination, and the while statement invariants generated by *fuzz-d* are therefore handled there, used to check program termination. As well as statements, *fuzz-d* achieves higher coverage over types, in particular following from its support of inductive datatypes: axioms are required in order to verify inductive datatypes with collection-type fields, and consequently higher coverage is achieved for `DataTypes`.

Analysing the weakness of *fuzz-d* over expression translation reveals that this difference also generally follows from the two tools supporting differing sets of language features. For example, both `ExpressionWelformed` and `ExpressionTranslator` contain considerable amounts of code relating to arrow-type expressions, which *fuzz-d* does not generate for. However, there are some instances missed by *fuzz-d* which likely result from the reduced Dafny subset it uses for verification, which it should ideally be able to cover. In particular, in `ExpressionWellformed` there is functionality for checking well-formedness of collection-type comprehensions which were coverable with XDsmith. Since *fuzz-d* currently omits this feature for verification, although it is capable of generating such expressions for compiler testing, this could suggest that *fuzz-d* may have unnecessarily over-reduced the size of the Dafny subset which it supports.

Evaluating *fuzz-d* against the Dafny Integration Tests

Overall, the verifier-oriented Dafny integration tests were able to achieve 31.98% higher line coverage than tests generated by *fuzz-d*, comprehensively covering a significantly larger proportion of language features, in particular more powerful verification features. These include ghost members, lemmas and predicates, and also more experimental verification features such as two-state functions which allow for comparison between two heap states, and witness clauses which are used alongside type definitions to define the default case such that the type is always non-empty. The coverage difference over verifier files is shown in Table 5.16.

File	Coverage Difference (%)	Lines Missed
DafnyCore/Verifier/Translator.BoogieFactory.cs	+37.48%	
DafnyCore/Verifier/Translator.ClassMembers.cs	+58.12%	
DafnyCore/Verifier/Translator.cs	+56.08%	25
DafnyCore/Verifier/Translator.DataTypes.cs	+29.82%	
DafnyCore/Verifier/Translator.ExpressionWellformed.cs	+58.10%	
DafnyCore/Verifier/Translator.TrStatement.cs	+62.86%	
DafnyCore/Verifier/Translator.ExpressionTranslator.cs	+40.74%	
DafnyCore/Verifier/ProofObligationDescription.cs	+39.39%	

Table 5.16: Comparison between fuzz-d (baseline) and the Dafny integration test suite over verifier testing coverage

Comparing the two results over the features supported by *fuzz-d* identified the same issue as XDsmith in that the language set supported for verification may have been over-reduced in

size – the integration tests are able to cover features related to collection-types, such as comprehensions, which arguably *fuzz-d* should be able to support. However, the tests also cover a lot of error cases which we should not necessarily expect *fuzz-d* to handle. For example, in `ProofObligationDescription` there are functions forming Boogie queries to check for missing pattern match cases, valid array initialisation size etc. Due to the nature of fuzz testing and the importance of program validity, *fuzz-d* doesn't naturally generate programs that can check Dafny handles these cases correctly, It is generally not the responsibility of *fuzz-d* to cover such error cases, although they do provide useful insight into how the mutator could further invalidate programs.

While the Dafny test suite was able to cover significantly more of the verifier-related code, there remain a number of lines which were coverable by *fuzz-d*, but not the tests – these are summarised by Table 5.16. Unfortunately, we are unable to provide detail into such cases as making interpretations on the verifier codebase requires levels of expertise beyond the requirements for this project.

5.5.3 Summary

In response to **RQ5**, we have shown that, even in its current limited form, *fuzz-d*'s verifier testing workflow is able to offer an improvement in coverage over XDsmith. Although we have so far not been able to identify any bugs using the mechanism, we have been able to find areas of the language that are not yet well-supported for verification, and which could be suggested as enhancements for future language versions. There is evidently still much more we could do to expand the verifier testing, shown by the difference in coverage to Dafny's regression tests; however, given that *fuzz-d* can increase the current random verifier testing standard with a potentially over-reduced language subset, there is clearly potential for this testing to further raise the bar once more language features are supported.

6 Conclusion and Future Work

In this project, we have introduced *fuzz-d* and demonstrated its ability to successfully test the Dafny compiler in an automated fashion, through randomly generating programs using a generational approach and performing differential testing to identify discrepancies over the backends. Alongside the core generation module of *fuzz-d*, we have implemented a number of additional modules. Firstly, we have effectively utilised the reconditioning approach to ensure validity of generated programs, introducing a novel extension to this – advanced reconditioning – which uses runtime information to limit the expressiveness constraints from applying reconditioning to programs. Additionally, we have developed an independent Dafny interpreter which resolves limitations in bug detection via differential testing solely over Dafny's backends (Section 3.4), and further integrate it into a workflow which generates annotated programs in order to identify soundness issues in the Dafny verifier.

From the results we have collected in the evaluation studies, it is evident that *fuzz-d* has effectively achieved its primary aim of identifying bugs within Dafny. So far, we have used *fuzz-d* to identify 35 issues (Section 5.1), resulting in 14 bug reports being made to Dafny developers on GitHub [38], of which five are confirmed and three are fixed. Moreover, *fuzz-d* has been shown to raise the standard for covering Dafny via compiler testing (Section 5.2), achieving over 13% higher line covering than XDsmith – the current compiler testing tool for Dafny – and notably covering over 50% of Dafny's core compiler files. Although the Dafny integration tests can cover a wider range of features than *fuzz-d*, we have been able to identify a number of weak spots in their coverage, in particular relating to missing tests for four binary operators. We further identified weaknesses in the Dafny integration tests via mutation testing, showing that there were some mutations which *fuzz-d* could kill but the Dafny tests could not. While the addition of advanced reconditioning and verifier testing did not lead to any additional bugs being found, they still provide additional value to *fuzz-d*, and there is much scope for increasing the value they add in the future.

In conclusion, our results demonstrate the success of testing campaigns using *fuzz-d* to identify weaknesses in Dafny, both in the form of bugs and through finding weaknesses in the integration test suite. However, Dafny is constantly evolving and therefore there remains more research which could be undertaken to more extensively test the programming language as new, experimental features are added.

6.1 Future Work

Although *fuzz-d* has proved successful in identifying bugs in Dafny, it is not without weaknesses, and we would like to consider making the following improvements in the future:

- **Increased Language Support**
  When comparing coverage of *fuzz-d* to Dafny's testsuite, its largest shortfall was in having less ability to test a diverse range of language features. In particular, while *fuzz-d* supports a range of different types, it still omits key type-system features such as newtypes, coinductive datatypes, arrow types and bitvectors. Implementing for these language features would be a priority in moving towards increased language support in *fuzz-d*.
- **More Powerful Annotations**
  Currently, the verifier testing performed by *fuzz-d* is limited in the amount of language

features it can support since a lot of them require additional specification mechanisms. Implementing heuristics to enable *fuzz-d* to annotate programs using these language features would help significantly improve the power of its verifier testing.

- **Automating Test Case Reduction**
  Throughout development and testing, the process of selecting programs triggering interesting behaviour, creating an interestingness test and passing them to a reducer was very much done manually. Naturally, there is a limit to the number of times this process can be repeated manually in a given time frame, as well as the manual approach simply being more error-prone, thus it would be extremely helpful to create an automated system which takes interesting programs and produces a reduced test case, possibly run alongside the testing campaigns in a producer-consumer workflow.
- **Investigating Dafny's Command-Line Options**
  Dafny's command-line interface offers a host of options, particularly corresponding to different optimisations which can be taken in the verifier, yet testing performed by *fuzz-d* currently utilises very few of these. It would be beneficial for *fuzz-d* to better utilise these options, since they may result in the internal ASTs being transformed in more interesting ways that could ultimately lead to more bugs being triggered.
- **Experimenting with Fuzzing Techniques**
  fuzz-d performs generation of test programs using a generational technique. This provides a good basis for other fuzzing techniques to build on – for example, we could investigate using equivalent transformations in a mutational approach (Section 2.2.2) to produce a number of different, but equivalent programs. These may exercise AST transformations in more diverse ways and consequently reveal latent bugs. In particular, it would be interesting to apply this to verifier testing, since Dafny or the SMT solver may perform different levels of optimisation based on the mutations applied.

6.2 Ethical Considerations

This project develops a tool which is used to help improve the reliability and stability of Dafny. Part of this process involves identifying bugs in the software it tests; reporting such bugs to the compiler developers could help achieve greater correctness in Dafny programs which are compiled using the software. However, bugs also represent weaknesses in a particular piece of software and malicious users attempt to exploit these in an attack. With compilers being such critical and widely-distributed pieces of software, there could be severe and large-scale consequences if a malicious user is able to identify and exploit a bug, therefore it is important that bug-finding tools such as *fuzz-d* are used appropriately and only for the benefit of the users of the software it tests. While there is risk of exploitable bugs being found without such tools being used, their existence makes this prospect far more likely.

When bugs are found using this tool, there involves an aspect of communication and collaboration in delivering the reports to the compiler developers such that they can be fixed. It is important that reports for critical bugs are delivered in a timely manner to minimise time taken to resolve the issue, but it would be bad practice and poor collaboration to continue to submit multiple bug reports in a short time period as this would overwhelm developers. It is important to validate bugs locally as much as possible prior to report submission, and submitting pull requests for smaller bugs might help maintain a positive relationship and ensure the developers are receptive to submissions.

A | Generated Code for Dafny Bugs

A.1 Forall Expression Inside Match Statement

```
1  static Main(__noArgsParameter) {
2        let _source0 = _module.D.create_A();
3        if (_source0.is_A) {
4          let _0_a;
5          let _init0 = function (_1_i1) {
6            return _1_i1;
7          };
8          let _nw0 = Array((new BigNumber(24)).toNumber());
9          for (let _i0_0 = 0; _i0_0 < new BigNumber(_nw0.length); _i0_0++) {
10           _nw0[_i0_0] = _init0(new BigNumber(_i0_0));
11         }
12         _0_a = _nw0;
13         for (const _guard_loop_0 of _dafny.IntegerRange(_dafny.ZERO, new BigNumber((_2_a
               ).length))) {
14           let _3_i2 = _guard_loop_0;
15           if ((true) ∧ (((_dafny.ZERO).isLessThanOrEqualTo(_3_i2)) ∧ ((_3_i2).
                 isLessThan(new BigNumber((_0_a).length))))) {
16             (_0_a)[(_3_i2)] = _3_i2;
17           }
18         }
19       } else {
20       }
21       return;
22 }
```

B | Summarised Coverage Results

DafnyCore File	XDSmith Avg. Line (%)	*fuzz-d* Avg. Line (%)	Dafny Tests Line (%)
DafnyOptions.cs	0.0	0.0	59.81
Rewriters/ConstructorWarning.cs	0.0	0.0	0.0
GeneratedFromDafny.cs	0.0	0.0	0.0
DooFile.cs	16.15	16.15	8.07
Compilers/Python/Compiler-python.cs	100.0	100.0	92.81
AlphaConvertingSubstituter.cs	0.0	0.0	13.33
AST/AstVisitor.cs	58.13	75.19	92.63
AST/BuiltIns.cs	92.63	91.05	100.0
AST/Cloner.cs	52.42	66.23	61.6
AST/DafnyAst.cs	46.15	46.15	85.29
AST/Expressions/ComprehensionExpr.cs	31.14	42.62	62.82
AST/Expressions/DisjunctivePattern.cs	0.0	0.0	88.46
AST/Expressions/Expressions.cs	22.87	31.12	71.52
AST/Expressions/ExtendedPattern.cs	0.0	47.82	36.95
AST/Expressions/IdPattern.cs	0.0	64.1	89.74
AST/Expressions/LetExpr.cs	45.83	50.0	87.15
AST/Expressions/LitPattern.cs	0.0	0.0	97.06
AST/Expressions/MemberSelectExpr.cs	40.66	56.0	85.66
AST/Expressions/NestedMatchCase.cs	0.0	100.0	100.0
AST/Expressions/NestedMatchCaseExpr.cs	0.0	94.11	94.11
AST/Expressions/NestedMatchCaseStmt.cs	0.0	82.35	82.35
AST/Expressions/NestedMatchExpr.cs	0.0	74.54	90.0
AST/Expressions/NestedMatchStmt.cs	0.0	80.51	81.81
AST/Expressions/OldExpr.cs	0.0	0.0	81.81
AST/Expressions/QuantifierExpr.cs	23.25	48.83	55.81
AST/Expressions/StaticReceiverExpr.cs	56.66	80.0	87.5
AST/Expressions/UnchangedExpr.cs	0.0	0.0	78.12
AST/ExtremeCloner.cs	0.0	0.0	63.15
AST/ExtremeLemmaBodyCloner.cs	0.0	0.0	83.63
AST/ExtremeLemmaSpecificationSubstituter.cs	0.0	0.0	81.63
AST/FreeVariablesUtil.cs	49.62	67.66	75.93
AST/Function.cs	49.8	51.35	68.24
AST/Grammar/IndentationFormatter.cs	0.0	0.0	0.0
AST/Grammar/ParseErrors.cs	100.0	100.0	100.0
AST/Grammar/Printer.cs	30.52	45.7	79.92
AST/Grammar/SourcePreprocessor.cs	37.73	37.73	18.86
AST/Grammar/TokenNewIndentCollector.cs	0.0	0.0	0.0
AST/Grammar/TriviaFormatterHelper.cs	0.0	0.0	0.0
AST/IncludeHandler.cs	84.61	84.61	100.0
AST/IteratorDecl.cs	0.0	0.0	52.4
AST/MemberDecls.cs	50.0	54.0	82.15
AST/Method.cs	54.14	51.38	68.09
AST/Statements/AlternativeLoopStmt.cs	0.0	0.0	77.57
AST/Statements/AlternativeStmt.cs	0.0	0.0	84.06
AST/Statements/AssertStmt.cs	0.0	0.0	64.88
AST/Statements/AssignOrReturnStmt.cs	0.0	0.0	35.78
AST/Statements/AssignStmt.cs	49.09	72.72	70.42

AST/Statements/AssignSuchThatStmt.cs	0.0	0.0	74.23
AST/Statements/AssumeStmt.cs	0.0	0.0	56.67
AST/Statements/BlockStmt.cs	22.22	22.22	22.22
AST/Statements/BreakStmt.cs	0.0	53.84	76.92
AST/Statements/CalcStmt.cs	0.0	0.0	65.33
AST/Statements/CallStmt.cs	46.15	80.76	96.15
AST/Statements/ConcreteUpdateStatement.cs	69.56	69.56	69.56
AST/Statements/DividedBlockStmt.cs	0.0	29.41	29.41
AST/Statements/ExpectStmt.cs	0.0	0.0	88.46
AST/Statements/ForallStmt.cs	0.0	67.27	62.34
AST/Statements/ForLoopStmt.cs	0.0	36.58	68.29
AST/Statements/IfStmt.cs	52.5	52.5	84.17
AST/Statements/LoopStmt.cs	0.0	52.38	80.47
AST/Statements/ModifyStmt.cs	0.0	0.0	76.85
AST/Statements/OneBodyLoopStmt.cs	0.0	21.56	60.13
AST/Statements/PredicateStmt.cs	0.0	0.0	54.54
AST/Statements/PrintStmt.cs	76.0	76.0	88.0
AST/Statements/ReturnStmt.cs	25.0	25.0	100.0
AST/Statements/Statements.cs	40.67	45.76	68.9
AST/Statements/UpdateStmt.cs	62.39	72.07	38.46
AST/Substituter.cs	33.49	51.63	86.56
AST/SubstitutingCloner.cs	0.0	100.0	100.0
AST/Tokens.cs	0.0	0.0	44.5
AST/TopLevelDeclarations.cs	67.81	67.81	68.38
AST/TupleTypeDecl.cs	98.11	94.33	100.0
AST/Types/ArrowType.cs	86.36	48.48	95.45
AST/Types/Types.cs	59.45	58.47	79.2
AST/VisibilityScope.cs	81.81	81.81	95.45
BigIntegerParser.cs	77.77	77.77	100.0
CompileNestedMatch/MatchAst.cs	0.0	35.48	52.1
CompileNestedMatch/MatchFlattener.cs	7.74	58.76	84.05
ConcreteSyntax/ConcreteSyntaxTree.cs	89.93	91.94	91.94
ConcreteSyntax/ConcreteSyntaxTreeUtils.cs	86.76	95.58	95.58
ConcreteSyntax/FileSyntax.cs	100.0	100.0	100.0
ConcreteSyntax/ICanRender.cs	100.0	100.0	100.0
ConcreteSyntax/LineSegment.cs	100.0	100.0	100.0
ConcreteSyntax/NewLine.cs	100.0	100.0	100.0
ConcreteSyntax/Verbatim.cs	100.0	100.0	100.0
DafnyConsolePrinter.cs	16.86	16.86	16.86
DafnyFile.cs	39.75	39.75	19.88
DafnyMain.cs	0.0	0.0	68.81
Feature.cs	0.0	0.0	76.31
Generic/BatchErrorReporter.cs	95.23	95.23	100.0
Generic/ErrorRegistry.cs	0.0	0.0	7.83
Generic/Name.cs	55.55	55.55	66.66
Generic/Node.cs	17.64	18.3	63.66
Generic/Reporting.cs	0.0	0.0	68.62
Generic/SccGraph.cs	58.43	63.0	73.78
Generic/SinglyLinkedList.cs	0.0	73.33	74.67
Generic/Util.cs	33.43	36.07	50.27
JsonDiagnostics.cs	0.0	0.0	40.0
LegacyUiForOption.cs	22.85	22.85	22.85
MergeOrdered.cs	0.0	0.0	0.0
Options/BoogieOptionBag.cs	61.03	61.03	61.03
Options/CommonOptionBag.cs	87.78	87.78	90.45

Options/DeveloperOptionBag.cs	93.75	93.75	93.75
Options/ICommandSpec.cs	97.05	97.05	97.05
Options/ProjectFile.cs	0.0	0.0	0.0
Plugin.cs	0.0	0.0	19.23
Resolver/Abstemious.cs	15.78	15.78	56.14
Resolver/BitvectorOptimization.cs	100.0	100.0	100.0
Resolver/BoundsDiscovery.cs	0.0	45.84	95.44
Resolver/NameResolutionAndTypeInference.cs	45.51	57.82	90.94
Resolver/Resolver.cs	29.91	38.35	72.33
Resolver/TypeInferenceChecker.cs	100.0	85.71	93.65
Resolver/CallGraphBuilder.cs	62.96	74.07	97.42
Resolver/ClonerButIVariablesAreKeptOnce.cs	0.0	0.0	0.0
Resolver/ExpectContracts.cs	0.0	0.0	0.0
Resolver/ExpressionTester.cs	45.07	49.01	91.0
Resolver/GhostInterestVisitor.cs	24.77	50.44	87.27
Resolver/InferDecreasesClause.cs	40.93	44.29	96.64
Resolver/PreType/PreType.cs	0.0	0.0	0.0
Resolver/PreType/PreTypeResolve.cs	0.0	0.0	0.0
Resolver/PreType/PreTypeToType.cs	0.0	0.0	0.0
Resolver/PreType/UnderspecificationDetector.cs	0.0	0.0	0.0
Resolver/PrintEffectEnforcement.cs	37.68	37.68	57.97
Resolver/ResolutionContext.cs	79.48	92.3	94.87
Resolver/ResolutionErrors.cs	100.0	100.0	100.0
Resolver/ResolverBottomUpVisitor.cs	100.0	100.0	100.0
Resolver/RunAllTestsMainMethod.cs	0.0	0.0	0.0
Resolver/Scope.cs	82.53	87.3	88.88
Resolver/SubsetConstraintGhostChecker.cs	40.62	54.68	79.55
Resolver/TailRecursion.cs	36.28	51.05	81.85
Resolver/TypeConstraint.cs	52.57	51.88	97.94
Rewriters/AutoContractsRewriter.cs	4.33	4.33	66.18
Rewriters/AutoGeneratedToken.cs	50.0	50.0	62.5
Rewriters/AutoReqFunctionRewriter.cs	5.35	5.35	5.35
Rewriters/ForallStmtRewriter.cs	100.0	100.0	91.63
Rewriters/IncludedLemmaBodyRemover.cs	75.0	75.0	100.0
Rewriters/InductionHeuristic.cs	0.0	0.0	86.86
Rewriters/InductionRewriter.cs	34.32	34.32	85.82
Rewriters/IRewriter.cs	95.0	95.0	95.0
Rewriters/JavadocLikeDocstringRewriter.cs	0.0	0.0	0.0
Rewriters/LocalLinter.cs	100.0	100.0	90.74
Rewriters/OpaqueMemberRewriter.cs	23.71	23.71	91.75
Rewriters/PluginRewriter.cs	0.0	0.0	0.0
Rewriters/PrecedenceLinter.cs	100.0	100.0	96.29
Rewriters/ProvideRevealAllRewriter.cs	19.04	19.04	100.0
Rewriters/QuantifierSplittingRewriter.cs	100.0	100.0	100.0
Rewriters/RefinementTransformer.cs	25.0	41.66	74.59
Rewriters/TimeLimitRewriter.cs	33.33	33.33	89.74
Rewriters/TriggerGeneratingRewriter.cs	76.92	100.0	100.0
TestGenerationOptions.cs	20.0	20.0	20.0
Triggers/ExprSubstituter.cs	0.0	0.0	82.69
UndisposableTextWriter.cs	0.0	0.0	0.0
Verifier/BoogieStmtListBuilder.cs	82.75	89.65	96.55
Verifier/CaptureStateExtensions.cs	31.57	31.57	31.57
Verifier/FreshIdGenerator.cs	100.0	100.0	100.0
Verifier/FunctionCallSubstituter.cs	0.0	0.0	92.0
Verifier/PrefixCallSubstituter.cs	0.0	0.0	100.0

Verifier/Translator.BoogieFactory.cs	60.61	68.83	86.23
Verifier/Translator.ClassMembers.cs	38.04	65.8	88.37
Verifier/Translator.cs	38.87	52.72	84.95
Verifier/Translator.DataTypes.cs	51.37	68.62	98.44
Verifier/Translator.ExpressionWellformed.cs	42.69	51.21	93.61
Verifier/Translator.TrStatement.cs	16.13	48.51	89.26
Verifier/Translator.ExpressionTranslator.cs	53.25	66.04	89.7
Parser.cs	34.54	43.7	33.36
Scanner.cs	64.13	64.13	69.16
Verifier/ProofObligationDescription.cs	0.0	0.0	30.23
Triggers/QuantifiersCollection.cs	0.0	100.0	97.75
Triggers/QuantifiersCollector.cs	42.55	78.72	100.0
Triggers/QuantifierSplitter.cs	28.26	67.39	92.66
Triggers/TriggerExtensions.cs	0.0	100.0	80.13
Triggers/TriggersCollector.cs	0.0	90.0	78.93
Triggers/TriggerUtils.cs	0.0	72.83	96.45
Plugins/DocstringRewriter.cs	0.0	0.0	0.0
Plugins/IExecutableBackend.cs	73.33	73.33	80.0
Plugins/PluginConfiguration.cs	54.54	54.54	54.54
Plugins/Rewriter.cs	0.0	0.0	0.0
Compilers/CompilerErrors.cs	0.0	0.0	0.0
Compilers/CoverageInstrumenter.cs	33.33	33.33	39.21
Compilers/Cplusplus/Compiler-cpp.cs	0.0	0.0	0.0
Compilers/Cplusplus/CppCompilerBackend.cs	7.5	7.5	10.0
Compilers/CSharp/Compiler-Csharp.cs	29.63	57.31	96.81
Compilers/CSharp/CsharpBackend.cs	65.45	59.09	66.36
Compilers/CSharp/Synthesizer-Csharp.cs	3.78	3.78	3.78
Compilers/Dafny/Compiler-dafny.cs	0.0	0.0	0.0
Compilers/Dafny/DafnyBackend.cs	12.0	12.0	16.0
Compilers/DatatypeWrapperEraser.cs	48.0	78.67	98.0
Compilers/ExecutableBackend.cs	49.59	49.59	50.4
Compilers/GoLang/Compiler-go.cs	34.63	58.91	91.06
Compilers/GoLang/GoBackend.cs	43.8	43.8	46.66
Compilers/InternalCompilersPluginConfiguration.cs	100.0	100.0	100.0
Compilers/Java/Compiler-java.cs	34.16	57.44	89.07
Compilers/Java/JavaBackend.cs	48.43	29.68	49.21
Compilers/JavaScript/Compiler-js.cs	30.36	51.32	88.27
Compilers/JavaScript/JavaScriptBackend.cs	69.81	69.81	73.58
Compilers/Library/LibraryBackend.cs	6.25	6.25	8.33
Compilers/Python/PythonBackend.cs	33.84	33.84	36.92
Compilers/SinglePassCompiler.cs	36.28	63.64	88.06
Auditor/Assumption.cs	0.0	0.0	0.0
Auditor/Auditor.cs	30.5	30.5	30.5
Auditor/AuditReport.cs	0.0	0.0	0.0

Table B.1: Summary of results from the compiler coverage experiment

Bibliography

[1] dafny-lang community. Dafny documentation: Dafny reference manual. URL http://dafny.org/dafny/DafnyRef/DafnyRef.html. Accessed: 20th December 2022.

[2] Microsoft. Dafny: A language and program verifier for functional correctness. URL https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/. Accessed: 20th December 2022.

[3] A. Irfan, S. Porncharoenwase, Z. Rakamaric, N. Rungta, and E. Torlak. Testing Dafny (experience paper). In *ISSTA 2022*, 2022. URL https://www.amazon.science/publications/testing-dafny-experience-paper.

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283– 294, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450306638. doi: 10.1145/1993498.1993532. URL https://doi.org/10.1145/1993498.1993532.

[5] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1), 2 2020. ISSN 0360-0300. doi: 10.1145/3363562. URL https://doi.org/10.1145/3363562.

[6] U. Schindler. The real story behind the Java 7 GA bugs affecting Apache Lucene / Solr. URL https://blog.thetaphi.de/2011/07/real-story-behind-java-7-ga-bugs.html. Accessed: 20th December 2022.

[7] S. Lee, H. Han, S.K. Cha, and S. Son. Montage: A neural network language Model-Guided JavaScript engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630. USENIX Association, 8 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/lee-suyoung.

[8] LLVM documentation: LLVM's analysis and transform passes. URL https://llvm.org/docs/Passes.html. Accessed: 28th December 2022.

[9] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107– 115, 7 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538814. URL https://doi.org/10.1145/1538788.1538814.

[10] V. Livinskii, D. Babokin, and J. Regehr. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.*, 4(OOPSLA), 11 2020. doi: 10.1145/3428264. URL https://doi.org/10.1145/3428264.

[11] K.V. Hanford. Automatic generation of test cases. *IBM Syst. J.*, 9:242– 257, 1970.

[12] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216– 226, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594334. URL https://doi.org/10.1145/2594291.2594334.

[13] A.F. Donaldson, H. Evrard, A. Lascu, and P. Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA), 10 2017. doi: 10.1145/3133917. URL https://doi.org/10.1145/3133917.

[14] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda. Random testing of C compilers targeting arithmetic optimization. 2012.

[15] E. Nagai, A. Hashimoto, and N. Ishiura. Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions. *IPSJ Transactions on System LSI Design Methodology*, 7:91– 100, 08 2014. doi: 10.2197/ipsjtsldm.7.91.

[16] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 78– 88, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450314541. doi: 10.1145/2338965.2336763. URL https://doi.org/10.1145/2338965.2336763.

[17] S. Veggalam, S. Rawat, I. Haller, and H. Bos. Ifuzzer:: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, volume 9878, pages 581– 601, 09 2016. ISBN 9783319457437. doi: 10.1007/978-3-319-45744-4_29.

[18] X. Liu, R. Prajapati, X. Li, and D. Wu. Reinforcement compiler fuzzing, 2019. URL https://openreview.net/forum?id=r1gCRtIDoE.

[19] C. Lindig. Random testing of C calling conventions. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, AADEBUG'05, pages 3– 12, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930507. doi: 10.1145/1085130.1085132. URL https://doi.org/10.1145/1085130.1085132.

[20] K. Even-Mendoza, C. Cadar, and A.F. Donaldson. Closer to the edge: Testing compilers more thoroughly by being less conservative about undefined behaviour. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pages 1219– 1223, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3418933. URL https://doi.org/10.1145/3324884.3418933.

[21] B. Lecoeur, H. Mohsin, and A.F. Donaldson. Program reconditioning: Avoiding undefined behaviour when finding and reducing compiler bugs. *Proceedings of the ACM Programming Languages*, 7(undefined), 2023.

[22] W.E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4):293– 298, 1978. doi: 10.1109/TSE.1978.231514.

[23] W.M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10:100– 107, 1998.

[24] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases, 1998. URL https://arxiv.org/abs/2002.12543.

[25] The LLVM compiler infrastructure. How to submit an llvm bug report, 2023. URL https://llvm.org/docs/HowToSubmitABug.html. Accessed: 14th January 2023.

[26] A.F. Donaldson, H. Evrard, and P. Thomson. Putting Randomized Compiler Testing into Production (Experience Report). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1– 22:29, Dagstuhl,

Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. ISBN 978-3-95977-154-2. doi: 10.4230/LIPIcs.ECOOP.2020.22. URL https://drops.dagstuhl.de/opus/volltexte/2020/13179.

[27] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 335– 346, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312059. doi: 10.1145/2254064.2254104. URL https://doi.org/10.1145/2254064.2254104.

[28] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 361– 371, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180236. URL https://doi.org/10.1145/3180155.3180236.

[29] J. Holmes, A. Groce, and M. Alipour. Mitigating (and exploiting) test reduction slippage. pages 66– 69, 11 2016. doi: 10.1145/2994291.2994301.

[30] K.R.M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348– 370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4.

[31] AWS Encryption SDK. URL https://github.com/aws/aws-encryption-sdk-dafny. Accessed: 2nd June 2023.

[32] J. Tristan. Personal Communication, 4 2023.

[33] K.R.M. Leino. This is Boogie 2. 6 2008. URL https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.

[34] I.T. Kassios. Dynamic frames and automated verification. 2011. URL https://pm.inf.ethz.ch/publications/Kassios11.pdf.

[35] W. Hatch, P. Darragh, and E. Eide. XSmith. URL https://docs.racket-lang.org/xsmith/index.html. Accessed: 23rd December 2022.

[36] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 65–76, 7 2015.

[37] S. Chaliasos, T. Sotiropoulos, D. Spinellis, A. Gervais, B. Livshits, and D. Mitropoulos. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 183–198, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523427. URL https://doi.org/10.1145/3519939.3523427.

[38] A. Usher. Dafny github issues by alex-usher, . URL https://github.com/dafny-lang/dafny/issues?q=is:issue+author:alex-usher+. Accessed: 25th May 2023.

[39] A. Usher. Js, python runtime exception: Forall inside match statement, . URL https://github.com/dafny-lang/dafny/issues/3987. Accessed: 25th May 2023.

[40] A. Usher. Java, C# and go back-ends: Errors using variables in match expressions, . URL https://github.com/dafny-lang/dafny/issues/3910. Accessed: 25th May 2023.

[41] A. Usher. C# backend: Runtime exceptions for modulus of large multiset, . URL https://github.com/dafny-lang/dafny/issues/3988. Accessed: 25th May 2023.

[42] F. Madge. Runtime implementation of multisets has a cardinality limit. URL https://github.com/dafny-lang/dafny/issues/4012. Accessed: 25th May 2023.

[43] Alex Usher. Java back end: incompatible type error (crash), . URL https://github.com/dafny-lang/dafny/issues/3854. Accessed: 25th May 2023.

[44] A. Usher. Internal translation error: while loop condition with sets/multisets, . URL https://github.com/dafny-lang/dafny/issues/3906. Accessed: 25th May 2023.

[45] Coverlet: Cross-platform code coverage. URL https://github.com/coverlet-coverage/coverlet. Accessed: 26th May 2023.

[46] Stryker Mutator. URL https://stryker-mutator.io/docs/stryker-net/introduction/. Accessed: 8th June 2023.