Imperial College London

Metamorphic Testing for Software Libraries and Graphics Compilers

Author: Andrei Lascu

Thesis submitted to

Department of Computing

Imperial College London

in partial fulfilment for the award of the degree of

Doctor of Philosophy

November 2021

Acknowledgements

I would like to thank my supervisor, Alastair Donaldson, for helping me throughout my degree, and polishing my researcher skills like a drip of water on a rock. I would like to thank Tobias Grosser, for acting like an unofficial co-supervisor, and for accepting to undertake the collaboration that eventually led to the work presented in this thesis. I would like to thank the research group at Imperial, who kept my sanity in check to not succumb to the pitfalls of the dark corners of the PhD. I would like to thank my family, for their continuous support through all this time.

Statement of Originality

I declare that the work in thesis is my own, with guidance from my supervisor and collaborators. All the text is my own, except where otherwise quoted, and references are included, as appropriate.

Andrei Lascu

Copyright Declaration

Abstract

Metamorphic Testing is a testing technique which mutates existing test cases in semantically equivalent forms, by making use of *metamorphic relations*, while avoiding the *oracle problem*. However, these required relations are not readily available for a given system under test. Defining effective metamorphic relations is difficult, and arguably the main obstacle towards adoption of metamorphic testing in production-level software development. One example application is testing graphics compilers, where the approximate and under-specified nature of the domain makes it hard to apply more traditional techniques.

We propose an approach with a lower barrier of entry to applying metamorphic testing for a software library. The user must still identify relations that hold over their particular library, but can do so within a development-like environment. We apply methods from the domains of metamorphic testing and fuzzing to produce complex test cases. We consider the user interaction a bonus, as they can control what parts of the target codebase is tested, potentially focusing on less-tested or critical sections of the codebase.

We implement our proposed approach in a tool, `MF++`, which synthesises `C++` test cases for a `C++` library, defined by user-provided ingredients. We applied `MF++` to 7 libraries in the domains of satisfiability modulo theories and Presburger arithmetic,. Our evaluation of `MF++` was able to identify 21 bugs in these tools.

We additionally provide an automatic reducer for tests generated by `MF++`, named `MF++R`. In addition to minimising tests exposing issues, `MF++R` can also be used to identify incorrect user-provided relations. Additionally, we investigate the combined use of `MF++` and `MF++R` in order to augment code coverage of library test suites. We assess the utility of this application by contributing 21 tests aimed at improving coverage across 3 libraries.

Contents

List of Figures

List of Tables

1 Introduction

Modern software engineering practice has evolved very quickly—from when patches to software would be sparsely deployed due to logistical difficulties, the advent of the internet made it possible to easily make patches and software updates available to the user. Nowadays, patches are deployed without users ever interacting with the respective software, in an automated fashion, and multiple patches a week, or even a day, is not unheard of. This accelerated software engineering system must be supported by solid systems that reduce the extent to which malformed or malign software is deployed to users. This is even more important for software such as compilers and software libraries: the former due to programs generally expecting the compiler will produce correct code, in line with the written program, and the latter which forms chains of dependencies across the software ecosystem, and a faulty link can affect vast swathes of dependent software. Thus, with continuous software development, there must be continuous testing to ensure correctness. Furthermore, as software evolves in size and logic, so must the testing techniques evolve to be able to handle these complex systems—not only to ensure that the software is free from functional bugs, but also free from *correctness bugs*: the software does what is expected of it to do, as designed by the logic behind the code.

There are multiple readily available techniques at the hand of developers to test their software for correctness: inline assertions, or unit tests being some examples. However, as the code-base increases in size, and more developers are working together each on their own section, it is seldom feasible for a developer to completely understand what the code itself does. As such, more advanced testing techniques are required. One such technique is *metamorphic testing*, which is more suited to checking semantics bugs in a given system under test. The idea is simple: for a given implementation of some operation, if we can define some relation R over a pair of inputs, and some expected relation S over the outputs of applying the operation over the inputs, then checking whether S holds gives confidence that the implementation is correct. These relations are generally defined by humans with knowledge of the respective domain the system under test resides. The relation between the inputs and the outputs comprises a *metamorphic relation* (or MR). For an intuitive example, we expect that adding 0 to some

integer input would yield equal outputs if common mathematical operations are applied.

While metamorphic testing is an inherently strong technique, providing semantics testing capabilities, employing it is by no means an easy endeavour. Finding MRs for a given system under test is not an easy task, requiring expert knowledge of the specific system, or being able to translate existing identities from compatible domains. The high upfront cost, and potential lack of expected results (the quality of the testing is heavily dependent on the quality of the MRs provided) might explain why metamorphic testing seems to not have seen adoption in industry. In this work, we explore the suitability of metamorphic testing as a helpful technique for finding semantics bugs across various computing domains, including satisfiability modulo theories libraries, Presburger arithmetic, and graphics compilers. Furthermore, we provide an infrastructure to make metamorphic testing more accessible to software developers, abstracting away part of the work required to employ metamorphic testing, while ensuring that developers keep control over what aspects of the respective system under test are exercised, in order to ensure that the testing process is meaningful.

In addition to exploring the bug-finding capabilities of metamorphic testing, we explore additional uses of *test-case reduction methods* in our work. By making use of the fact that metamorphic testing comes with an in-built oracle which is able to internally evaluate the correctness of a test, we can then attempt to employ test-case reduction to simplify a test with respect to a specific property of interest. While usually test-case reduction is used to simplify a failing test-case, removing parts of the given test which are not required in order to reach the failure state, we can perform reduction with respect to some other property of interest. One such example would be reducing with respect to covering some code of interest in the system under test.

1.1 Contributions

We describe a practicality-focused variation of metamorphic testing, which we name *metamorphic testing with high-level operations*, removing some of the theoretical burden of formally defining MRs (Chapter 3). With some user-provided ingredients which interface with the a desired software library to be tested, we employ methods from the domain of metamorphic testing and fuzzing to synthesise fresh test cases. The intuition behind the approach is finding semantically equivalent, but syntactically distinct implementations provided by some software library (more generally, finding various ways of reaching the same result in different ways, based on what a software library provides). The core ingredient that the user must provide are equiv-

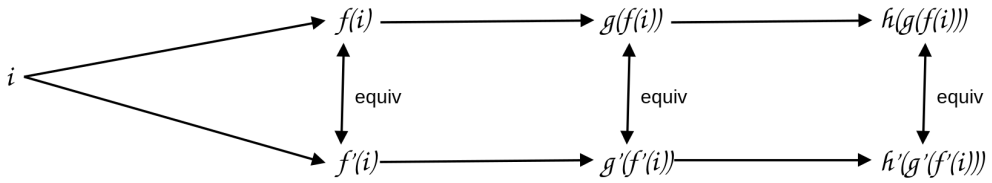


Figure 1.1: Abstract view of metamorphic test with high-level operations

alent implementations which implement observably equivalent functionality of some high-level operation. To exemplify, suppose we have a library which implements file-system operations. One possible functionality, or *high-level operation*, is moving files on the disk. The library can of course provide a `move` function which provides this, but the same result can be observed if we `copy` the source file to the destination, then `remove` the original copy.

Having multiple such high-level operations, each with a set of possible implementations, allow us to create chains of MRs via *composition of metamorphic relations* [47] (which can be regarded as a single, expressive MR), by having an abstract chain of high-level operations, which can be instantiated to a concrete implementation. This is visualised in Figure 1.1. From an initial input i, we create two chains of high-level operations, with each operation being concretized by some randomly chosen implementation: the first operation concretized by f and f', the second by g and g', and the third and final operation by h and h'. As each implementation should result in the same observable functionality, they are expected to be pair-wise equivalent, based on that functionality (e.g., taking the above `move` example, this means we'd observe the original file present at the destination, but not at the source).

We regard this approach as a novel example of *metamorphic fuzzing*. To support the application of metamorphic testing with high-level operations, we propose a test synthesis infrastructure, where the user is tasked to provide details about the API of the library under test, which is then used to randomly generate test cases to be practically executed. We propose such an implementation, targeting C++ APIs of software libraries, with a tool called `MF++`. We discuss how, while our evaluation is practically focused on our implementation, it is generally applicable to any programming language and domain, and what requirements are required.

We present an in-depth case study, applying our technique and implementation to 7 mature and widely-used libraries (Chapter 4). We provide details on how the user-provided ingredients, named *specifications*, are written, and how `MF++` was designed to ensure a familiar interface for developers to write these specifications. To further help developers, we identify methods of abstracting specifications for a domain of interest. By identifying common operations that a library within a domain is expected to implement, we can write MRs with respect to an

abstract implementation of these common operations. A specific API can then make use of `MF++` testing by linking the abstract operations with concrete API calls for their library. We evaluate our technique on two fronts. Primarily, we focus on the ability of our tool to find bugs in these libraries. This effort lead to finding 21 across 4 tested libraries, all of which have been reported and subsequently fixed by the respective maintainers. Second, we investigate the ability to cover additional code that the test suites of these libraries do not cover, as a measure to safeguard against potential future bugs. Additionally, we focus on the usability of our tool, considering an expert user employing it for their library, and the effort required to apply our tool.

We discuss an implementation of an automatic custom test-case reducer for `MF++` tests, named `MF++R` (Chapter 5). For randomly generated test cases, test reduction is an important step, in order to identify the core issue that triggers a fault in the underlying system under test. In the case of `MF++`, the need for reduction is doubled as an approach to identifying incorrect user-provided MRs. Our implementation performs reductions that maintain program validity, and is able to remove large parts of unneeded code automatically. As an extension, we employ the notion of *cause reduction* [34] to reduce test cases with respect to code coverage achieved which is not provided by test suites of libraries under test. We use this approach to contribute 21 coverage enhancing tests to test suites of 3 open-source libraries.

We evaluate the effectiveness of a custom implementation of automatic test case reduction, comparing the implementation time against the usefulness of general-purpose reduction tools applied to highly specific domains. We further employ our custom reducer in to augment code coverage achieved by existing testing suites of the libraries we test.

Finally, we present details of a separate project which applies metamorphic testing over the domain of graphics shader compilers, which are a key component of modern graphics drivers (Chapter 6). Our tool, `GraphicsFuzz`, demonstrated the possibility of mixing metamorphic testing (albeit limited to functionality-preserving MRs) and fuzzing in order to effectively find bugs. This project laid the groundwork for the `MF++` project, and subsequent work around `MF++`. We discuss a number of functional and security defects identified by applying `GraphicsFuzz` to consumer products.

Each chapter includes relevant related work, but we provide some general background in Chapter 2. We present ideas and future work in Chapter 7.

To further detail the contributions in this thesis, we aim to answer the following questions:

1. Is metamorphic testing with high-level operations effective at finding bugs?

2. Can metamorphic testing with high-level operations be used to improve existing coverage metrics?

3. Could metamorphic testing with high-level operations be applied to generic domains?

1.2 Publications

Aspects of this work have been included in the following publications:

- A submission to the International Conference on Software Testing, Verification and Validation (ICST) 2022 conference;
- A submission [43] to the Metamorphic Testing (MET) 2021 workshop, receiving the best paper award;
- A submission [26] to the Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) 2017 conference;
- A short paper submission [28] to the MET 2016 workshop.

2 Background

In this section, we will provide some background details about the testing techniques which were used as basis to develop the work presented in this thesis.

2.1 Oracle Problem

In software testing, an *oracle* represents a mechanism which, for a given test-case, is able to determine whether that test-case is expected to fail or pass. The *oracle problem* [6] is a well known and prevalent issue which testing techniques must face. The problem states that for a **given input** to a system, there is no method of programmatically determining the correct output. Generally, assessing whether an execution or output is correct is something done with human knowledge, who can use their knowledge of the system. An automated testing method must provide a way to either leverage this human knowledge and include it in tests for the system, or to somehow derive it. Of course, as systems grow more complex, the existence of an oracle becomes less certain, and perhaps even intractable or outright impossible [77].

Nevertheless, systems must be given a degree of confidence that they are correct, even without checking for full functional correctness. We distinguish a number of such oracles, and emphasise their potency:

Implicit oracles Oracles with very weak potency. These would include runtime checks provided by the language in which a computer program is encoded, for example bounds check present in Java. These checks are agnostic in relation to the semantics of the program, and only offer some rudimentary protection again well known and common programming issues. These checks can be augmented by the use of third party tools, such as `Address Sanitizer` [70] or `Valgrind` [57], which perform different categories or checks, but increase confidence that no underlying coding issues are present.

Annotation-based oracles Oracles with rather weak potency, but dependent on user thoroughness. These are assertions written by the programmer, that ensure the state of the

program has certain properties at certain points in the code. They are likely a consequence of the program implementation, but are generally constrained in their checks both in terms of strength and scope, in relation to the entirety of the program.

Manual oracle Very strong oracle, but requiring an investment of both human knowledge and time, making them impossible to scale. They represent an expectation of specific program functionality, and are generally encoded as post-condition checks within individual test cases: some input is provided to the whole or part of the system under test, then properties over the outputs are checked against. These properties are determined manually, with knowledge of the functionality of the SUT, as well as the domain in which it resides.

With human knowledge as the main driver behind providing oracles, automatically overcoming the oracle problem is an issue of great importance. There are some methods which attempt to either generate explicit oracles, or somehow work around the problem in specific ways. Some examples:

Metamorphic Testing This technique is described in more detail in Section 2.2.

Differential Testing In this method, there is an assumption that multiple implementations of the desired functionality exist. For example, there are multiple libraries available for networking. When sending a file over a network, no matter what particular library of choice we use to connect to the network, we expect the same file to arrive at the destination. More generally, if there is some functionality provided by multiple different implementations, we can cross-check that the expected result is identical no matter the chosen implementation. We note that this technique requires a deterministic domain to test against.

Tests with expected results This approach involves randomly generating test cases in such a way that after generation, the outcome of the test case is already known. An example of using this approach is, during generation, keeping a table of expected values for generated variables, which allows the values to be checked at execution time if they match the expected, generation time values [48, 55].

Historical oracles In the case of `EvoSuite` [31], the tool generates unit tests which assert the state of the system after the application of some randomly generated operations. By itself, this does not constitute an oracle, and requires manual inspection to ensure the state is consistent with expectation of the tested system. However, applying the test

suite generated by `EvoSuite` across different versions of the system under test can identify inconsistencies where the state of the system has changed due to changes in the source code of the system. Of course, the issue with this approach is that these changes might be an expected consequence of the internal code changes, and not representative of critical features of the underlying system, due to the random nature of `EvoSuite`.

2.2 Metamorphic Testing

Metamorphic Testing [19, 68] is a testing technique based on relations between inputs to an SUT, and expected relations between outputs, while overcoming the oracle problem.

The classic example of a metamorphic relation is defined in terms of the *sine* function. Supposing we have an implementation of *sine* we would like to check. Then, by definition of *sine* in mathematics, it must be the case that for some input x, we have that $\sin(x) = \sin(x+\pi)$. Another example can be considered in the domain of graphs. Supposing we implement an algorithm P to return the shortest path between two vertices (a, b) in an undirected graph G. If we find that $P(G, a, b) \neq P(G, b, a)$, then our algorithm is clearly faulty.

Formally, an MR is defined as follows:

> Suppose there are two inputs (x, y), two binary relations (R, S), and a system under test defined as a function f. If (R, S) is a metamorphic relation and $(x, y) \in R$, then $(f(x), f(y)) \in S$.

There are two observations to be made in the above example. The MR does not specify anything about the *correctness* of the implementation in any way, but simply checks that the property we are looking for holds. Therefore, an implementation of the graph algorithm above which would always return a constant value, say 0, would not be marked as incorrect by the above check in isolation. Of course, additional checks can give more confidence that the implementation does something according to expectations, but this is better checked via a suite of hand-written tests (which metamorphic testing should be used to augment, not replace). For example, we might check that $a \neq b \rightarrow P(G, a, b) > 0$. This issue is due to the fact that metamorphic testing is applied to *implementations*, rather than *semantics*.

Another observation is that the MR itself contains an oracle, defining whether the test is expected to fail or pass. Using the MR $P(G, a, b) = P(G, b, a)$, we can select any two vertices in G and expect the MR to hold, as it is a tautology. Further, applying any changes to G, such as adding additional points, or moving points, should not affect the validity of the

MR. Additionally, we can use certain MRs to generate additional input. Taking the MR $\sin(x) = \sin(x + \pi)$ means that for any test case which contains an instance of $\sin(x)$, we can generate a new test case by replacing such an instance with $\sin(x+\pi)$, and the expected output would be maintained.

2.2.1 Generating Metamorphic Relations

As the core component of metamorphic testing, MRs are important both to get right and ensure that meaningful checks are performed. This usually requires in-depth knowledge of both the domain, and the SUT over which the MR is applied to. Previous work on metamorphic testing is usually focused on few, high-quality MRs (as highlighted in Section 6.4 of the Metamorphic Testing Survey [68]).

More generally, sources of inspiration for MRs can be derived from other domains, and translated to the domain of the SUT. For instance, boolean identities can be translated to the domain of arithmetic sets. An example is $A \wedge \perp = \perp$ is equivalent to $S \cap \emptyset = \emptyset$. Other, more specific MRs, could be translated across domains, or serve as inspiration. In addition, there are proposed techniques to use existing MRs as basis for more complex MRs. One such example is *composition of MRs* [47], which is akin to chaining compatible MRs together, thus increasing the search space, therefore potency, of one MR application. Another source of MRs is identifying *redundancy* in software [15]: by identifying two distinct parts of the source code which can perform the same semantic action (e.g., as exemplified, a **putAll** function called once versus a **put** function called multiple times), any divergence in observed execution indicates a potential bug in one of the implementations. While not explicitly considered as such, we believe this approach is an instance of metamorphic testing. With the recent advent of machine learning, there are also techniques for attempting to automatically synthesise MRs for respective SUTs [39].

When writing MRs, we might also be interested in whether certain MRs might be more useful in finding bugs than others. One study [21] evaluates this over both black-box (no knowledge of the specific SUT, but domain knowledge), and white-box testing (knowledge of both the specific SUT and the domain). Their results indicate that, for black-box testing, what might theoretically seem like a strong MR over, might not be as effective in practice than other MRs. Furthermore, for white-box testing, attempting to generate "different" executions (in terms of paths traversed, sequence of executed statements, etc.) via the MRs did indeed prove more fruitful. A question for the latter result is how scalable this is: supposing we have a monolithic

million-line SUT, how would we identify whether two executions are different enough. One possible solution is using code coverage as a metric. The follow-up question is then, is this information sufficient for a human to be able to come up with effective MRs?

Another approach is to somehow quantify the strength of MRs, and to use such metrics to drive the definition of future MRs [14, 37]. Various statistical approaches are used to quantify the bug-finding capabilities of MRs based on the difference with other MRs. More specifically, supposing we have two MRs, and associated bug-finding metric. If we can define some relation over the two MRs, such as difference in achieved code coverage, or frequency in exercising a certain branch, then we can create a profile of how differences in the effect of an MR (in terms of exercise parts of the SUT) can give rise to bug-finding capabilities. For future potential MRs, we could then compare them with existing MRs, in order to estimate a potential bug-finding capability.

2.2.2 Applying Metamorphic Testing

There are a number of techniques which we believe are instances of metamorphic testing, but have not been explicitly noted as such.

Equivalence modulo inputs [44] is a compiler testing technique which first profiles test cases over some input I to identify dead code (i.e., code not executed during runtime) for that input, then alters these dead code sections in order to attempt to trigger compiler bugs. We argue that this is an instance of metamorphic testing, by using the relation of equality over outputs, and "identical live code sections" over inputs.

Another approach is *semantic fusion* [78]. This technique works in the domain of satisfiability modulo theories (SMT) [8], and takes two equisatisfiable (i.e., eitehr both satisfiable or unsatisfiable) SMT formulas as input. It then essentially fuses the two formulas together, via various steps, such as concatenating the two formulas together, or creating fresh variables defined based on existing variables in either formula. Due to the equisatisfiability of the original formulas, an instance of a fused formula is expected to have the same satisfiability, while the fusing process should maintain well-definedness. Thus, multiple fused formulas can be created, and their satisfiabilities checked, with a mismatch indicating a potential bug in the solver itself. We believe this to be an instance of metamorphic testing, as **(a)** it takes existing input and modifies it in a way that a fresh input is produced, and **(b)** the expected result of the fresh input is predetermined by construction. It uses the two inputs to produce a new test case, with known satisfiability based on the two input tests, and intermingling control flow from the

two inputs. This method can also be considered an instance of metamorphic testing, where the satisfiability of the inputs affect the satisfiability of the test formula. Furthermore, there is a relation to be defined between the new test case and the two input test formulas, in terms of syntax.

The above two techniques are more practical applications of metamorphic testing. Traditionally, metamorphic testing is applied more theoretically. For instance, metamorphic testing could be applied to programs with random outputs, by statistically modelling the output, based on the input [36]. An implementation of this proposed approach is heavily dependent on the specific use case, as the way the system is affected by random input would determine how the MRs relating outputs are defined. Therefore, there can be no one particular, practical implementation of this proposed method.

2.2.3 Metamorphic Testing and Differential Testing

Differential Testing [52] is a technique where, if there are multiple implementations of some system available (e.g., multiple compilers for a given language), then we can take a valid input, execute it over all available systems, and check results. Assuming the executions are deterministic, not having a single, consistent result across all systems indicates a potential bug. A different approach to differential testing is *n-version programming* [18], which involves multiple systems being created for a specification, and ensuring all systems are consistent with one another. The main difference with differential testing is that instead of finding these distinct systems, they are instead created.

Metamorphic testing can be considered similar: it functions at a similar level of redundancy, but instead testing redundancy **within** a given system, rather than **across** different ones. That is where one of the main distinctions lie: while metamorphic testing can work within the confines of a single implementation, by definition, differential testing involves the existence of two equivalent systems (in practice, this could potentially be the same system with different parameters, such as one compiler with different optimisation levels being expected to produce functionally equivalent binaries), while in n-version programming, there is the additional cost of building the systems as well.

For the purposes of software library testing, which this works focuses on, we believe metamorphic testing is more suitable, as it is unlikely to find two libraries in the same domain taking identical inputs; a library usually has its own API it consumes. Thus, differential testing would be difficult to apply, but not impossible — one could consider building a generic fuzzer, which

produces input programs based on some API specification.

2.3 Fuzzing

Generating random tests in order to trigger crash bugs is known as *fuzzing* [54]. The concept is fairly simple: software should be able to handle any input, including invalid or unexpected input, and not crash, but gracefully terminate. Fuzzers can be generally applicable, and a tool like american fuzzy lop (AFL) [81], which works by fuzzing random bit-sequences, simply fuzzing domain agnostic input, looking for instances of program crashes. The more domain knowledge we take into consideration, the more engineering work, but the more potentially useful bugs are found. For instance, a compiler might be less interested in a crash involving a sequence of non-UTF-8 characters, but might be interested if a string containing rarely used characters causes the whole compiler to crash. There is a very fine balance to consider between fuzzing meaningful tests, writing a very specific fuzzer that is not reusable, and targeting bugs that matter.

For the consideration of what are bugs that matter, it mostly depends on the point of view with which we inspect the system under test. For instance, from a security stand-point, crashes can potentially be used as part of a complex attack vector, to exploit other security flaws and perform some malicious acts on the underlying system. However, a more common perspective is that of functionality, whether the system under test behaves in a manner consistent with expectations. It is hard to define what these expectations are exclusively at the implementation level. If some documentation is provided, we can potentially use that to indicate whether some observable execution is correct, but this is hard to encode within a fuzzer.

Fuzzing as a testing technique is rather easy to apply. However, engineering the syntax and perhaps some semantics (e.g., if we are testing a tool which includes statically sized containers, we do not want to access the container outside its boundaries) of the underlying tools is not trivial. This generally leads to having fuzzers which are extremely specialised for the particular tool or domain they are applied to. Furthermore, as the original intent of the technique, fuzzed tests generally do not come with a specific oracle: the oracle is whether the program crashes or not. Nevertheless, fuzzed tests can trigger functional assertion failures within tested tools.

There is a wide variety of available fuzzers in the testing ecosystem [45]. Based on their functionality, we can derive a number of categories for how a fuzzer performs [58, 1]:

Generation technique Fuzzers can either generate test cases from scratch, or use some given input to mutate into fresh, generally more expressive, tests;

Structure Whether the data generated by the fuzzer is aware of the expected input structure of the system under test—fuzzers are either dumb (unstructured) or smart (structured);

System knowledge Based on how much knowledge of semantics of the system under test the fuzzer is aware of in order to guide its fuzzing; from least knowledgeable, we distinguish black-box fuzzing, grey-box fuzzing, and white-box fuzzing.

Generally, fuzzers can be defined using these three categories to express what kind of tests they produce at the high level. Thus, we can define `AFL` [81] as a black-box, dumb, generator fuzzer, while `Orange4` [56] is a black-box, smart, mutating fuzzer. White-box fuzzer are more involved and rarer, and they usually include some sort of feedback mechanism to guide the fuzzing process. Examples would include *concolic execution* [32, 69] tools, such as `SAGE` [20].

2.3.1 Generation Methods

We distinguish two types of fuzzers: mutation-based fuzzers, and generation-based fuzzers. Mutation-based fuzzers, such as `libFuzzer` [2], or `afl` [81], take some pre-existing tests as input, which they then mutate in order to explore more of the SUT. The mutation is based on algorithms implemented in the fuzzer, potentially based on structure of the SUT. For instance, mutating an input `C` could involve editing an operator in an expression, which would require being aware of what an expression is, and how to identify its components. Generation based fuzzers, such as `Csmith` [80], or `CLsmith` [46], must be aware of the syntax (and, at some level, semantics) of the underlying SUT. For instance, to fuzz a `C` program, in the case of `Csmith`, the fuzzer must know the correct syntax to define a variable, or how to declare a function. Further, it must know that if it uses a function, it must have a definition, or else it would generate an invalid program, which would fail compilation. However, more subtle generation issues could appear, based on how much of the underlying SUT we test. For instance, if we would like to test arrays in `C`, then we need to consider array bounds, or using pointers as arrays. This makes generation-based fuzzers more difficult to implement.

One final aspect to mention when fuzzing is the need to ensure whatever is being fuzzed is *well-defined* with respect to the SUT. Generating an input which has undefined behaviour is likely to indicate a potential bug has been triggered, when actually the presence of undefined behaviour puts the execution in an invalid state. By interpolation, we can assume that all tests generated by such a fuzzer can be considered invalid, due to the potential presence of undefined behaviour.

2.3.2 Main Caveats

When employing fuzzing, while it is an effective and simple technique to apply in theory, there are a number of limitations the method suffers [10].

The first one is the oracle problem. Fuzzers are good at finding crash bugs, as such bugs are obvious: we can determine whether a program crashes by simply executing it. And, if our fuzzer is correctly implemented, then it should emit valid tests which are not expected to crash. However, could fuzzing be used to find more subtle bugs? One approach is to combine fuzzing with differential testing [80, 60].

One other aspect is when fuzzing is applied to a particular SUT, it has to be done by a specific implementation. That implementation might have certain properties which might lead to the SUT overfitting its implementation based on the types of test the fuzzer produces.

A third issue is *bug slippage* [22]: when a test is executed and a fault is observed, we do not know whether that test might expose any additional faults, were execution to proceed beyond the initial fault point. This phenomenon is important to keep in mind when reducing test cases: a naive implementation that reduces a test "while it fails" would potentially reduce away the original bug that was found, and uncover a second, hidden bug. This also indicates that fuzizng should be a continuous process, as fixing prior bugs might allow new ones to be uncovered by a fuzzer.

2.4 Automated Test-case Reduction

In the context of randomly generated test cases, automatic test case reduction is a valuable step. Randomly generated test cases are generally large, in order to be more likely to trigger bugs. Furthermore, due to their random nature, they are not expected to be readable by a human being. However, once a bug is triggered, a human must identify the problematic behaviour in the test case. Generally, it is only a small sequence within the test case which is required to trigger the bug. While a human can manually try to extract this require code, it is generally time consuming to do so.

Test case reduction can be performed in various ways [3], but there are two main ideas to explore. The first approach is reducing at the level of source code. The most direct method to this is performing string manipulation on the source code itself. The result of the initial, failing test is first logged. Then, parts of the source code are removed. This can be done at the level of instructions, function declarations, operations, but must be done in such a way to

maintain the syntactic validity of the program. In addition, during this process, it must be checked that the validity of the test is maintained. The most prominent issue to be taken care of is undefined behaviour. For example, the declaration of a variable can be reduced, leading to uninitialised memory being read. This approach is known as *delta debugging* [82].

An example of delta debugging is the `C-Reduce` [63] infrastructure. While it primarily targets `Csmith` [80] test cases, it is generally applicable with appropriate infrastructure adaptations to `C`-like languages. Generally, the tool works as described above, taking test cases as input and applying reductions via sub-string deletion at the source code level. It also validates the reductions by using third party tools to ensure the reduced test cases are well-defined and free from undefined behaviour. `C-Reduce` uses the concept of *interestingness* tests [24], where a user-provided code evaluates the reduced test case in order to determine whether it is interesting or not. This could vary from a specific return code being observed, or a specific output being emitted by the test.

2.4.1 Approaches to Reduction

While delta-debugging is quite widely used in order to reduce test cases, there are other approaches, which require some cooperation between a generator and a reducer, but are able to more effectively perform reduction due to the additional available metadata.

One such example of source-level reduction applicable to random test generators is encoding each random choice in the source code of the generated test case, alongside a pre-determined "minimal" value for that random choice. To exemplify, suppose we would like to generate the value `true`, but instead, we generate a boolean expression that is known to be true by construction, let's say `3 == f(3)`, where `f` is known to return a value equal to its parameter. In the source code of the generated test, we could then generate `GENERATED(3 == f(3), true)`, where `GENERATED` is a macro taking two parameters, and returning one of them, based on some flag. A reducer could identify such a pattern, and know that the whole instruction `GENERATED(3 == f(3), true)` can be minimized to the second argument, by construction. Such an approach allows the reducer to already "know" where it can apply reductions, as well as the minimal form the reduction can take. This method was used in `GraphicsFuzz` [26].

An alternative approach to reduction of randomly generated programs is to target the random choices themselves. A randomly generated program could be abstracted as a sequence of the choices made whenever randomness is invoked. Thus, if we log all random choices, then replay the generation process, but instead of providing random values whenever the random

generator is invoked, we provide previously recorded values, we can generate the same program exactly. Modifying the sequence of random choices should also modify the resulting program, and a "smaller" sequence of random choices could likely lead to a smaller program, or large observable reductions with little cost. This approach is implemented by `Hypothesis` [50], where the random choices made during generation of a test case are encoded in a bit-stream, and reductions are performed over this abstract bit-stream represenation of a program in order to minimize tests [49].

2.4.2 Scope of Reduction

Normally, test case reduction is applied on inputs which trigger some fault on an SUT, particularly when the input has been generated as part of an automated testing infrastructure. However, reduction could be used to target other specific properties of inputs. The idea of *cause reduction* [34] suggests defining some measurable property (in the usual case, it is a binary "does this program fail" check), and attempting to reduce the program while that property holds. One such example is reducing with respect to coverage. Suppose we have a test which exercise some specific part of code that was not covered by our existing test suite. We might want to augment our test suite with a test to include that additional coverage, or even understand why the existing tests were insufficient in triggering that coverage. Thus, our property of interest would be whether a particular instruction is covered by the reduced variant. This process would of course involve adding support to gather coverage during the reduction process, so slightly more complicated than simply executing the test (which checks itself whether it fails or not).

Coverage is not the only property of interest we could be reducing towards. We could consider performance, such as reducing necessary precision when representing floating point values [66], or performance of symbolic execution tools [83]. Full test suite reduction can be performed [65], such as finding duplicate tests functionality via dynamic inspection [53], or via machine learning methods [79].

Finally, one interesting point to note is that a reducer tool could also potentially be used as a fuzzer [62], in a metamorphic fuzzing fashion. The operations we would apply to our input would be to remove code. The idea is that, even under invalid input, some program should terminate gracefully, instead of crashing, or triggering an internal assertion. Of course, the utility of such found issues are questionable, but it can nevertheless be a good, and cheap source of further testing.

2.5 Graphics Compilers

Related to Chapter 6, we present the mechanism behind graphics compilers for the **Open Graphics Library** (OpenGL) [1] language.

OpenGL provides a cross-platform API for graphics processing, usually executed by a graphics processing unit (GPU). While not providing an implementation of its own, it provides a standard which describes in minute detail how an implementation is expected to behave, and a related conformance test suite to evaluate whether a given implementation is sufficiently compatible with the standard. In order to make use of OpenGL, there are a number of components that need to be specified. First, the **host code** represents code that is to run on the host CPU. This generally involves moving any data required in the memory of the GPU, including the code to be executed on the GPU, as well as calling the procedure to execute that code, and any clean-up required. The other components are called **shaders**, with one shader being required for each applicable **pipeline stage** [2]. For example, the **vertex shader** defines the object in space to be rendered, while the **fragment shader** defines how the rendered object is painted.

GPU vendors usually certify that their devices are OpenGL compliant, by subjecting their devices to the conformance test suite. However, what is interesting from a compiler testing point of view is that each GPU must then provide its own OpenGL compiler implementation. This compiler is included in the software driver of the GPU. This potentially gives rise to a large space of compilers, based on combinations of vendor, GPU device, driver version, and OpenGL version.

Another consideration of OpenGL in the context of testing is that it heavily involves usage of floating point values, which means inexact operations and potentially distinct rounding modes across configurations, and that the OpenGL standard is fairly loose in multiple places, allowing GPU vendors to make their own implementation-defined decision, while still conforming to the standard. These properties mean that testing approaches like differential testing are not directly applicable, as different configurations are not constrained to a single correct value. Therefore, testing methods which apply within the *same* test system are preferable.

[1] `https://www.opengl.org/`, accessed 28th of October 2021
[2] `https://opengl-notes.readthedocs.io/en/latest/topics/intro/opengl-pipeline.html`, accessed 28th of October 2021

3 Metamorphic Testing for C++ Software Libraries

While metamorphic testing is widely researched in the academic domain (as exemplified by the number of scientific publications focusing on it [68]), there is not much evidence of it being applied in industry. Based on our experience, we believe this to be due to the requirements to applying metamorphic testing, namely well-specified relations between the inputs and expected outputs. In the real world, where software is very rarely specified, defining expressive and correct relations could pose too large an initial hurdle to overcome.

Thus, we present *metamorphic testing with high-level operations*, which makes defining relations more practical. Furthermore, we present an implementation of our approach in the `MF++` tool, which targets `C++` software libraries. In the rest of this chapter we will provide an overview of metamorphic testing with high-level operations (Section 3.2), describe more details about the features and generation process of the approach (Section 3.3), and finalise with technical details about our implementation to provide a generic interface to `C++` software libraries to make use of our approach (Section 3.4).

3.1 Motivation

This project initially started with a simple requirement: try to apply metamorphic testing to the integer set library [73] (`isl`). Due to the nature of the library, namely implementing operations over sets, we believe metamorphic testing to be a suitable testing technique to apply to the library. This is due to the numerical nature of the library, allowing us to apply multiple mathematical identities, as well as the fact that certain features of the library are known to be under-tested (primarily, the *coalesce* [74] operation). For the latter, automatically generating input with metamorphic testing ensures that input is meaningful, as it can be derived from real-world examples. Eventually, we started exploring options to provide a more generic framework to apply metamorphic testing, which evolved into the current incarnation of `MF++`.

As presented in Section 2.2, the core of applying metamorphic testing (MT) is comprised of metamorphic relations (MRs). In the context of a simple operation, such as the exemplified *sine* operation, it is fairly obvious what kind of MRs we can consider, further helped by knowledge of the *sine* operation and related identities. We can go further, and say we could find MRs for common Linux commands, such as word count (first split the file at some whitespace character, apply word count over each partition, then sum the results). However, when talking about a complex system, such as a compiler, defining these MRs becomes a herculean task. Taking the entire program source code as input, how can we define a transformation to an equivalent source code, and further, how could we even begin to relate the expected binary output?

One approach would be to partition the input: can we find localised transformations which modify the program behaviour in some way, then extract, via some analysis, potential execution changes based on these transformations? Also, we probably prefer executing the program, rather than performing binary inspection. The main issue is there are some practical obstacles to performing traditional metamorphic testing to certain systems. Not to mention the need for specific knowledge regarding the underlying system and potentially the domain it resides in (which would be a large boon to producing MRs). Overall, for a programmer, both understanding metamorphic testing, and being able to write *expressive* MRs poses a difficult undertaking, with potentially high upfront cost.

3.2 Metamorphic Testing with High Level Operations

We remind the reader the main ingredient required to perform metamorphic testing of some given system under test (SUT) are MRs — for two inputs related by some relation R, if we can define some relation S over the outputs, then we can construct alternative inputs by using R to transform existing input data. Applying the system over these inputs, then checking that S holds over the newly produced output and the original, known output, provides a means of testing the SUT. Generating the MRs, however, is not an easy task. The most common way to derive them is to carefully inspect the formal specification, and using the knowledge of how the respective system under test affects a given input, derive expected outputs, as well as the second part of an MR, the expected relation between the obtained outputs.

Thus, we approached metamorphic testing with practicality in mind — how can we apply the technique in a manner that might be potentially not as effective as creating bespoke MRs for a given system under test, but generic and practical to allow for easy use of metamorphic testing? Additionally, the effectiveness could be partly augmented by throughput: if we have

a method of automatically generating tests that employ metamorphic testing, it means we can continuously generate tests, with the potential of finding bugs and exercising the SUT.

We devised with what we refer to as *metamorphic testing with high-level operations*. The core idea is that for a given SUT, we can define a set of *high-level operations* that the SUT can be made to implement. Some of these operations might be due to the domain the SUT operates in. Further, for each high-level operation, we define a set of distinct implementations, called *alternative implementations*, which, given the same input, would emit the same result. In layperson's terms, this can be thought of as doing the same thing in different ways. Suppose we would like to implement an algorithm computing the shortest path in an undirected graph. We expect to get the same result whether we compute the path from node a to node b, or vice-versa, or if we introduce a new node in the graph. Additionally, if we remove a node outright, we expect to compute a path shorter than or equal in length to the original value; similarly, an operation that might *fuse* a number of nodes (distinct from a and b) in some manner (for example, placing a new node at coordinates equal to the average of the input nodes), our algorithm should also return a value less than or equal to the original one. This could be considered a *removal* high-level operation. In terms of traditional metamorphic testing, this can be thought of as restricting the S relation to *equivalence*, and defining an implicit R by using internal instructions. In our proposed approach, by setting S to always be equivalent, we can simplify away one difficult aspect of defining MRs, essentially folding it into the declaration of alternative implementations. The main advantage is then one of practical usability; an expert of an SUT can more easily come up with some transformation of interest (which need not be complex), then think of additional ways, within the scope of the library, of how the transformation might be expressed in terms of other, existing SUT operations.

Even with this level of abstraction, generating MRs is still not entirely trivial. However, existing code can be used to generate MRs. Unit or regression tests, which are rather common place in industry projects, can provide a source of abstract high-level operations. Suppose the developer of the aforementioned shortest-path algorithm already had a test suite which included examples where nodes were removed, in order to check the output was updated appropriately on whether the node was on the originally computed shortest path or not. These examples can translate to alternative implementations of the `removal` operation described above. We note that an *operation* does not necessarily need to be something common-place. Of course, existing operations from the domain which the system under test implements are useful starting points (e.g., for a system under test implementing real integers, arithmetic identities are good sources of high-level operations). But more specific, SUT-level operations could be implemented. An

$A \cup A$	$A \cup \emptyset$
$A \cap A$	$A \cap \mathbb{U}$
$\overline{\overline{A}}$	$simplify(A)$

(a) Example alternative implementations for the *identity* high-level operation, using known mathematical set identities

$A \cup B$
$\overline{\overline{A} \cap \overline{B}}$
$union\_demorgan(A, B)$

(b) Example alternative implementations for the *union* operation

Table 3.1: Example of alternative implementations for a set library

example is an SUT which provides a highly optimised implementation for commonly used sequences of operations (akin to the floating point fused-multiply-add operation). Alternatively, it might be the case that some sequence of internal operations (potentially present in the regression test suite), not necessarily exposed to the end-user, might require further testing. We can summarise this aspect of metamorphic testing with high-level operations as follows: effective testing for a *particular* SUT is heavily specific to that SUT. Our approach provides a middle ground between crafting a highly specific test suite, and the strengths of metamorphic testing. This leads to an initial overhead, required to identify high-level operations and respective alternative implementations, but can lead to automatic and specialised testing.

To better illustrate metamorphic testing with high-level operations, we provide a more complete practical example. Suppose we have an implementation of a set arithmetic library. Further suppose our library implements the following operations: `union`, `intersection`, `complement`, `union_demorgan`, and `simplify`. The first three operations correspond to the same operations in mathematical set theory. The operation `union_demorgan` is a specialised implementation of the following identity derived from the DeMorgan laws: $A \cup B \equiv \overline{\overline{A} \cap \overline{B}}$. Finally, `simplify` is an operation which optimises the internal representation of a set, without affecting the observable contents of the set.

The first step involves identifying high-level operations of interest. Inspiration from the underlying domain, in this case mathematical set theory, is a good place to start. Thus, we can identify *union*, *intersection*, and *complement* as high-level operations, due to their correspondence to operations in set arithmetic. For simplicity, we will refer to the implementations of these operations by their corresponding mathematical symbols. Next, we observe that there is a library-specific internal operation, namely `simplify`. The (informal) specification of this operation says that the observable contents of the set should not change. We can therefore consider the operation an *identity* operation. This will constitute the fourth high-level operation for this exercise. We note that *identity* can always be considered a high-level operation in any library.

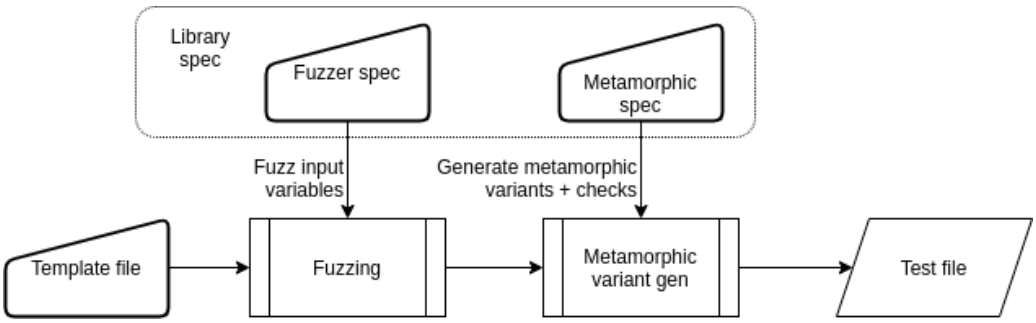


Figure 3.1: Workflow of metamorphic testing with high-level operations. The trapezoid shapes represent user-provided input, while the double-sided rectangles represent transformations

Next, we must look at potential alternative implementations. Similar to above, we start by using known set identities. Thus, for *identity*, as can be seen in Table 3.1a, we can directly use known identities with the corresponding internal implementations. Note that we can also consider the `simplify` operation as an instance of *identity*. To exemplify alternative implementations of *union*, consider Table 3.1b. Observe how there are two implementations, one implementing the DeMorgan law for union explicitly, using library operations, and one using the optimised `union_demorgan` implementation. While obviously these two should be equivalent, as we expect `union_demorgan` to be based on the explicit DeMorgan law, it is important to note that we are testing the actual implementations, and it is very possible for bugs to appear at any point in the code. Thus, even obvious equivalences should be tested.

3.3 Approach Overview

In this section, we provide a more in-depth overview of metamorphic testing with high-level operations. We describe the user-provided ingredients required to interface with the library under test, and discuss ways in which we augment the metamorphic testing process beyond the concept of applying MRs. Some of the design choices are referenced within the context of our implementation focusing on `C++` software libraries, `MF++`, but these should be considered as an example of the underlying principles, which are generally applicable.

The overall workflow of the approach can be seen in Figure 3.1. A user-provided *template file* is used as input, which links all the components together. This is augmented via fuzzing (Section 3.3.2), by using the *fuzzer specification* to generate variables, which are used as input to the metamorphic variant generation process. Finally, using the *metamorphic specification*, we generate metamorphic variants, as well as user-defined checks over the generated variants, to ensure that the variants are equivalent (Section 3.3.3).

3.3.1 User-provided Input

In this section, we shall discuss what kind of information is required by our approach in order to link with a specific SUT. This covers information required to perform fuzzing, generate metamorphic variants, check whether the generated variants are as expected, and a template to link all the components together.

Fuzzing Data

This specifies the types of the SUT that should be exercised, as well as functions which operate over those types, and constructors for the types. We expect that a library has a variety of internal API types, and some of them are core to the functionality of the SUT. For example, in a set arithmetic library, there conceivably exists a type representing a set, and a variety of functions representing operations over sets, and fuzzing API traces exercising these functions should prove interesting.

We note that this information could potentially be automatically parsed from the source code of the SUT. However, we consider that allowing the user to define the types and functions of interest serves two main purposes:

- It allows for a sort of in-built semantics to be declared over what kind of operations are visible to the fuzzer. There might be internal functions operating over types of interest which might not make sense to combine. For example, a function might translate the constraints of the function into the rational domain, while there exist functions which can only consume integer-domain sets. Ensuring that these two do not mingle in a completely random fashion ensures valid tests are generated.

- Targeted testing is possible by restricting what is visible to the fuzzer to desired sections of the codebase. For instance, there might be some logging functions which are less interesting when performing functional testing, or a newly-introduced feature might be more desirable to test.

MR Data

This is equivalent to defining MRs in traditional metamorphic testing. In our approach, declaring MRs involves two steps: identify a high-level operation that the SUT can conceivably perform, and then provide different implementations which achieve the same observable result.

We already saw some examples in Table 3.1. Test suites could potentially used for inspiration to identifying core functionality, and derive high-level operations and implementations from.

In addition to the alternative implementations, we consider metamorphic checks as a part of this data. These checks should ensure that, after variants have been generated, they are in an equivalent state. Most simply, two variants could be checked for equality using the API of the SUT. There could also be more specific tests. Going back to the arithmetic set example, we could check that the cardinality of the result sets is equal, or that some arbitrary value is either contained within both checked sets, or neither. These checks are the mechanism by which functional SUT bugs are caught.

Test Template

This represents the "glue" that holds everything together, and includes any SUT-specific setup or cleanup that might be required. For example, it might be the case that an SUT uses a `context` object to refer to within all its variable declarations. It is expected that we do not want to randomly create a variety of such contexts, but rather a single context for the entirety of the test case. Therefore, the template might contain some code to ensure that a singular context object is created. Subsequent operations should be able to identify this object as in scope, and know that it can be safely used when required.

Additionally, it might be the case that additional operations are desired to be done over fuzzed variables. There is potential of integrating code within this template which affects eventually generated objects. We view this template as the generating starting point, presenting an abstract view of what the eventual test case should look like, containing abstract operations which are expanded into applications of fuzzing and metamorphic testing.

3.3.2 Fuzzing

The main motivation behind implementing fuzzing is to have a never-ending supply of interesting input to apply metamorphic testing over. This ensures that there is no hard-limit on the supply of inputs to define MRs over — if more are needed, they are simply fuzzed automatically. Furthermore, it guarantees that inputs exist to build a test around; it might be the case that certain libraries might have insufficient publicly available examples to use as inputs. Finally, fuzzing might produce interesting inputs, triggering certain faulty edge states, which might not be easily triggered in day to day use. While the practicality of such tests might be questionable, from a testing point of view, it is at least useful to be made aware of the existence

set
set_int

(a) Example function types

Return Type	Function Name	Parameters
set	universe	[]
set	union	[set, set]
set	intersect	[set, set]
set	lt_cond_set	[set_int, set_int]
set	lte_cond_set	[set_int, set_int]
set_int	convert	[int]
set_int	neg	[set_int]
set_int	add	[set_int, set_int]

(b) Example function signatures

Table 3.2: Example input data for MF++ fuzzer

of these faulty edge cases, as they might be more easily triggered as software is developed.

Our implementation of fuzzing in MF++ follows a type-oriented, bottom-up approach. First, the fuzzer is initialised with a list of library-specific types to be used, as well as a list of function signatures to be used during the fuzzing process. We particularly distinguish the need to provide *constructor* functions for certain object types, which can be used to construct an object of the respective type either without any dependencies, or with dependencies that are assumed to be fulfilled. Further details on how this information is provided can be found in Section 3.3.1.

When the fuzzer is called, it is provided with the type of the object to be fuzzed. We distinguish three possible choices on how to construct such an object:

Provide an existing object In this instance, if there is an existing object of the required type in scope, the fuzzer might simply choose to return that object. This ensures that the control flow is not simply a straight line, and has some complexities (currently, the fuzzer does not produces any control flow cycles).

Invoke a non-constructor function Select a user-provided library function which returns an object of the requested type, and build an appropriate invocation, further fuzzing the necessary input.

Invoke a constructor function We know that constructor functions, by definition, can be used at any point. Constructor functions are distinguished by the fact that they do not require additional input be generated for them: either the required input is expected to be defined and in scope, or there is no input (e.g., a constructor for an integer-like variable could call the constructor with a randomly generated variable).

For an example, assume we have the data presented in Table 3.2, which shows some types of

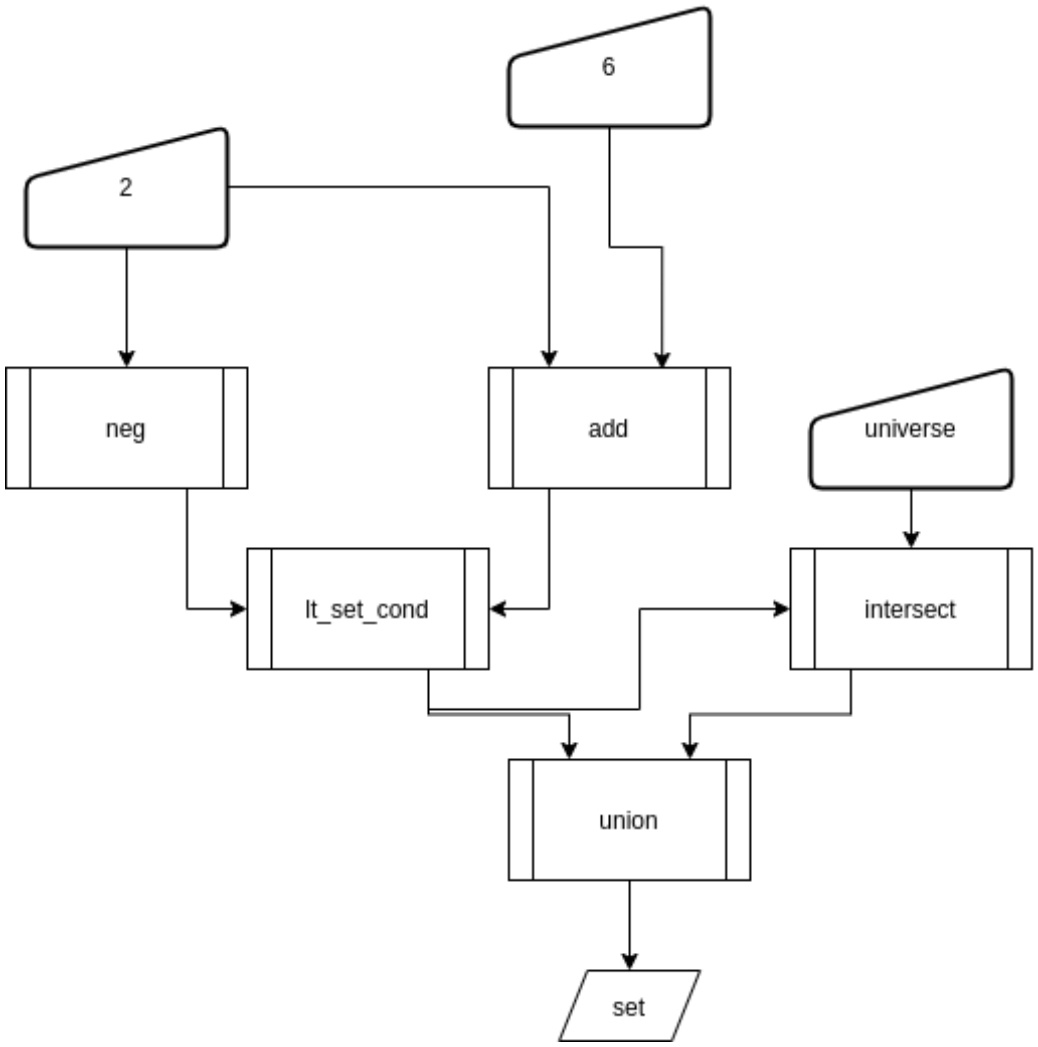


Figure 3.2: Example fuzzing process of a set in our example set library

interest from our set implementation library (Table 3.2a), as well as some function signatures of interest (Table 3.2b). There are two types of interest, `set` representing a mathematical set, and `set_int` representing a mathematical integer. We also have a number of functions from our library at our disposal during the fuzzing process. We distinguish some of them. Functions `universe` and `convert` are able to generate library objects without any dependency. The former simply creates a `set` on demand, while the latter requires an `int` as a parameter, which is a primitive type that can be randomly generated on the fly. These two are instances of what we consider *constructor* functions. Furthermore, we also distinguish the functions `lt_cond_set` and `lte_cond_set`. Semantically, these functions will create a set with the constraint that the provided parameters are less than (respectively, less than or equal) one another. Having a variety of types available as parameters in the provided functions means the fuzzer is able to exercise those types, increasing the testing space.

Now let's suppose we would like to fuzz an object of type `set`. A high level overview of a potential choice to produce such an object can be seen in Figure 3.2. The parallelogram

represents the output object (an object of the `set` type), double-lined rectangles represent function applications (as defined by the fuzzer specification), and trapeziums represent either terminal inputs (such as the integers `2` and `6`), or constructor functions (which will henceforth be considered terminal inputs for brevity). We observe that the left half of the figure indicates how, from two integer inputs, we use SUT functions to generate required `set` intermediate objects, via `lt_set_cond` to be used further in the fuzzing process. Note how the figure represents a directed acyclic graph, with all top nodes representing terminal inputs, and a sole bottom node representing the expected output object. Further note how objects can be reused across function calls: the output of `lt_set_cond` is used in both the sub-sequence `union` call, as well as the parallel `intersect` call. While not presented in this diagram, it might be the case that an object is chosen multiple times as a parameter to a fuzzed function invocation. The only restriction placed by the fuzzer when choosing what concrete objects to substitute for parameter requirements is that the types match.

One other potential idea to very finely control the fuzzing process is to add individual weights for each function available to the fuzzer. This would ensure that more interesting functions can be selected to be more likely produced, which is especially useful when there is a large choice space for a given object type. While from an engineering point of view, this is not a difficult task to implement, the more challenging issue is how to define these weights in a simple manner, without overloading the initial required overhead of specifying the library information.

3.3.3 Metamorphic Variant Generation

This section will discuss how the metamorphic variants are generated, which is the core component of `MF++`, as well as the focus of our application of metamorphic testing. We aim to generate metamorphic variants with a large number of calls to the API of the SUT, in order to more likely trigger bugs (as discussed in Section 3.3 of the Csmith paper [80]). At a high-level, this step will create a number of objects of a specified type, which are expected to be semantically equivalent by construction, and then perform pairwise checks between these variants. Assuming MRs are correctly specified by the user, a check failing would indicate a potential silent bug in the SUT (i.e., a bug which is not directly obvious, as opposed to a full crash or assertion failure). Similar to the fuzzing process, this step requires some user-provided input, in the form of the MRs to be used and the checks to be done between variants.

Following our approach of metamorphic testing with high-level operations, we categorise the MRs provided by the user in the following way: **(a)** each MR is associated with one operation;

(b) each operation contains at least one *base* MR, which doesn't recursively call other operations (and is, ideally, the simplest implementation for that operation); **(c)** and each operation is either a *first-class operation*, or a *second-class operation* (discussed in Paragraph 3.4.1). In order to apply our metamorphic testing with high-level operations idea, we first generate a *sequence* of operations (which essentially implements *composition of metamorphic relations* [47], in order to increase the number of SUT API calls), choosing randomly from among the existing, defined operations. We go back to our running mathematical set library, where we decided upon four operations: *identity*, *union*, *intersection*, and *complement*. We first set a length of our expected sequence, let's say three operations. This would yield a potential $4^3 = 64$ permutations just for the sequence of operations, as any of the operations can be chosen in any position (this is due to the fact that all of these operations are first-class operations, as will be discussed later). At just this point, we observe how rich the search space is: if we define n high-level operations, and decide upon a sequence length of m, there are n^m total possibilities (this makes generating similar sequences mathematically unlikely). Let's set our chosen sequence of operations as *union-identity-union*. Let's also set the number of expected metamorphic variants to three as well. We also assume we have three input objects of type `set` at our disposal, i_1 through i_3. During actual testing, the sequence length, and the number of variants to produce are provided as parameters to `MF++`, and we derive them empirically, trying to balance larger (thus more expressive) tests, with acceptable runtimes and timeout ratio.

With these ingredients, and the alternative, or *concrete*, implementations from Table 3.1, we can produce concrete variants using the API of our set library. The process is identical for each variant, and is visualized in Figure 3.3. We have three input variables to start with, (i_1, i_2, i_3). To generate one metamorphic variant, we select concrete implementations for each high-level operation in the sequence (noted by the labels at the top of the figure). We start with the first *union* operation. From the list of all MRs associated with *union*, we randomly choose one. For instance, we choose the simple concrete implementation of directly applying our library implementation of set union. Note that all concrete implementations of union take two `set` objects as inputs, and return another `set` object. For this step, we shall use two of the existing input variables as parameters to our choice of concrete operation, say i_1 and i_2. That will conclude concretizing the first high-level operation for the first variant. For the second operation, we must take into consideration that we have already performed some computation via the first concretization step. Therefore, one of the inputs to the eventual concrete implementation of the second (and subsequent operations) will have to be the current value of the partially computed metamorphic variant (represented by variable v_0 in the figure,

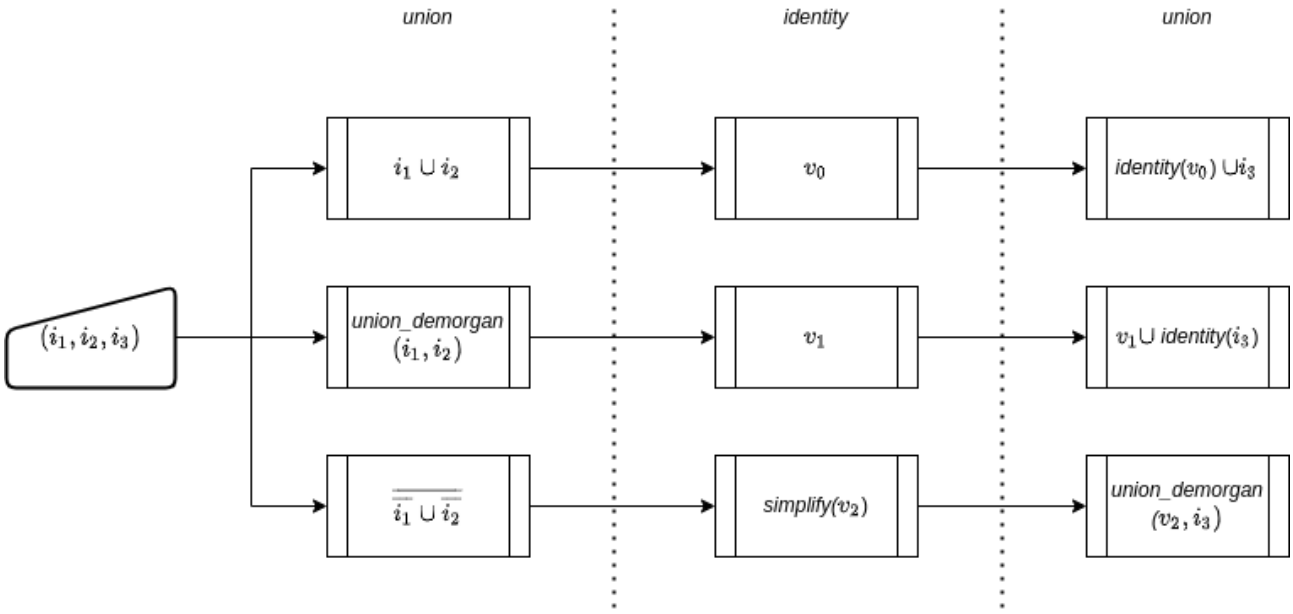


Figure 3.3: Example generation of three metamorphic variants with an operation sequence length of three. Input variables are represented by (i_1, i_2, i_3). The labels at the top indicate the high-level operation in the sequence. Each rectangle represents a concrete implementation for the respective variant of the high-level operation. Variables (v_0, v_1, v_2) represent the value returned by the previous concrete implementation in the sequence.

where zero indicates the index of the current metamorphic variant). This will always raise the issue of limitations in the signatures of these MRs (discussed in more detail in Section 3.4.4). To continue, we now must concretize an *identity* operation. Suppose we choose to concretize it via just returning the given value (i.e., we don't actually apply any operation). Finally, for the next concretization step, of a *union* operation, we shall choose to again apply our implementation of union. One parameter is set in stone, as it must be the existing value of our metamorphic variant. For the other parameter, we can again refer to our input variables, and choose an appropriate one from the given set of i_1 to i_3. Suppose we chose i_3 this time. Furthermore, we observe that we could choose to apply an *identity* operation to any of the parameters. By definition, we don't expect *identity* to change the semantic of the input, so it should be safe to apply *identity* at our discretion. Other variants follow the same logic to be generated.

Now let us discuss some of the finer details of this generation process.

Input variables The metamorphic test generator is safe to assume that it will always have the inputs that it needs. This is due to the fact that we constrain that parameters passed to an MR be distinct, unlike what can happen during the fuzzing process. That is, when parameters must be chosen to create a concrete MR invocation, we can imagine existing input variables as a pool, and we simply pick up one input from the whole pool. Theoretically, there is not

too much need for this particular requirement — we could have a single object of the correct type fit into all required slots. However, practically, if we consider that we might want to use operations such as *subtract*, it would be much more likely to render all computation trivial, by simplifying to some literal value (although an argument could be made that such a literal value might actually have some interesting internal properties, even if it appears trivial on the outside). In `MF++`, we manually ensure that the number of inputs is equal to the highest required number of inputs across all defined MRs, as we simply fuzz all the inputs.

Recursive MRs We briefly mentioned how it could be possible to insert an *identity* operation anywhere, as it should not change the semantics of the given input. This is a specific instance of what we call *recursive MRs*. More generally, we can replace any operation with a corresponding high-level operation, provided one exists. As *identity* is equivalent to a no-op, it is the case that it can be inserted anywhere. But, for example, if we consider the concrete implementation of *union* consisting of $\overline{\overline{A} \cap \overline{B}}$, we can see that there are applications of *complement* and *intersect* operations. We could replace these direct applications of the operations with our alternative implementations, which would potentially also, in turn, instantiate to MRs which call other high-level operations, and so on. This not only allows us to create much more complex sequences of operations, but also allows us to make use of second-class MRs.

Checks The checks are user-provided functions which are meant to check, in a pair-wise fashion, that generated variants are equivalent. We use equivalence, as that can be highly customisable based on the SUT itself, and potentially based on what kind of data is presented for use by the fuzzer and metamorphic variant generation. An example of a fairly common equivalence check, generally applicable to the majority of domains, is equality. However, a potentially interesting specification for our set library might enforce that the cardinality of all sets is a multiple of some number. Thus we could check variants respect that property after generation. In regards to how the checks are performed, at the end of the metamorphic variant generation process, we apply each check function in order between the currently produced metamorphic variant and the first produced metamorphic variant. This will ensure that the variables are fully generated and in scope. Furthermore, we assume the equivalence checks are transitive, therefore checking all variants against the first variants should be equivalent to checking across all variants.

A small note here on what "equivalence" might mean. We, as developers of the technique, do not enforce anything regarding the meaning of equivalence. This should be implicitly integrated

throughout the specification. Our approach is that the fuzzer and the metamorphic variant generation should maintain some observable property of interest, and the check should ensure that that particular property is maintained. As long as that particular property holds across variants, then the variants can be considered equivalent. Thus, we leave this to the developer, to decide which properties are of interest, and how to integrate this into the specification. An operation named *double*, where implementations might double either the count or value of elements, is perfectly reasonable, if the property we expect to be consistent across variants is not related to the absolute count or value of elements contained within a set.

These are all components of our implementation of metamorphic testing with high-level operations within `MF++`. The core idea refers to how we categorise MRs within operations, which allows us to create syntactically distinct and semantically equivalent objects by choosing different implementations for an operation. The randomness is further amplified by having a sequence of operations (akin to composition of MRs [47]), having recursion within MRs, and by introducing a fuzzing component for inputs.

3.4 Implementation Details

This section covers some more technical details of `MF++`, our implementation of metamorphic testing with high-level operations for `C++` software libraries. We also give an idea of how `Clang`[1] is used, particularly heavy usage of the `Clang` AST, in order to parse user-provided input, and generate explicit, valid test cases. We note that the ability to use the `Clang` AST is due to targetting `C++` software libraries, which is supported by the `Clang` tool. We note that the possibility of using source code to the extent it is done in `MF++` is possible due to the existence of `Clang`, and a similar implementation in another language would have to either find an appropriate parser, which exposes sufficient information, or provide a custom implementation that does so.

3.4.1 Parsing User-provided Input

This section discusses how user-provided input is parsed specifically by `MF++` to be used during generation, as well as the expected structure of this input.

[1] `https://clang.llvm.org/`, accessed 27th of January 2022

Library Specification

We have mentioned the notion of a library specification multiple times throughout. It is the core component of the provided user data in MF++, and consists of multiple files in the current incarnation of MF++. The specification is provided in native C++ code, made possible due to the use of Clang libtooling to implement MF++. Over time, the specification underwent multiple design choices, including being split up in multiple files in the latest iteration, for compartmentalization. We shall discuss each component individually, while keeping in mind that they are a part of the whole.

Fuzzer specification The first component we discuss is the information to be provided for the fuzzing element of MF++. This includes functions available for use during the fuzzing process, and internal types for the SUT.

MF++ allows for these to be defined at the level of the source code of the SUT, via attributes. As an initial proof of concept, due to custom attributes requiring modifying the source of the compiler used, to our knowledge, we have made use of the `annotation` attribute. This attribute takes a string parameter, which is exposed in the Clang AST upon parsing, allowing MF++ to identify the node for which the attribute was declared. Thus, we set a specific keyword in MF++, namely `expose`, and we log nodes of interest annotated with `__attribute__((annotate("expose")))`. Having access to the specific Clang AST node, we can log any information required by the fuzzer, such as the signature of the function exposed (i.e., return type, parameter types, parameter count), or the name of the type used. We note that this action is performed by the `libSpecReader` action (Section 3.4.2).

For our purposes, we create a duplicate of the original header files of the library under test, which we include in the template file. Passing the template file to our libtooling implementation will also parse any included header files, allowing our attributes to be available in the Clang AST. In order to fulfil dependencies, as the parsing process must be given a fully correct C++ file, we make use of `compile_commands.json`, as described in the Clang documentation, pointing to a valid installation of the respective library under test[2]. It is probably good practice to have a separate test header for MF++, however, we believe our choice of annotation should not clash with any other tool. Furthermore, future updates might allow for a custom annotation to be included in Clang, ensuring no clashes.

In addition to the source code annotation, we also parse exposed types defined within the `fuzz` namespace, and functions defined within the `fuzz::lib_helper_funcs` namespace. The

[2] `https://clang.llvm.org/docs/HowToSetupToolingForLLVM.html`, accessed on 24th of August, 2021

Listing 3.1: Example of some potential checks for set library

```
1 void is_equal(set s1, set s2) {
2   assert(is_equal(s1, s2));
3 }
4 void is_equal_count(set s1, set s2) {
5   assert(count(s1) == count(s2));
6 }
```

former functionality is particularly useful to mask certain internal types (e.g., the user might want to distinguish between sets with rational elements and sets with integer elements, but there is only an internal `set` type available to present both), or to `typedef` more complex types that might not be handled by `MF++` due to not parsing those specific AST nodes (i.e., we have not tested `MF++` on variadic templates). The latter functionality can similarly allow users to definer a coarser granularity on the functions allowed the be used by the fuzzer. For example, we might want to check that the divisor of a division is non-zero, or that the exponent to a power operation is positive, if we would like to ensure we keep all numerical values in the integer domain.

Metamorphic specification This is the most important component of the tool, and where most of the work to polish the eventually generated test cases should go into. The metamorphic specification contains the definition of the high-level operations, both first- and second-order, and the checks to be performed. The specifications make use of namespaces in order to distinguish these categories of functions, both for readability, and for internal use by `MF++`.

All this information is expected to be in a top-level namespace named `metalib`. Then, we distinguish three further namespaces: `checks`, `generators`, and `relations`. These correspond to the checks, the second-class MRs, and first-class MRs respectively. We now discuss each of them in part, with examples given for our on-going set library example.

The `checks` namespace should be comprised of function definitions which embody correctness checks over the metamorphic variants. There is no homogenised signature requirement over the checks, but we expect there to be at least one parameter of the type of the metamorphic variants, in order to allow the check to be performed over the respective metamorphic variant it is generated for. Normally, we'd expect two such parameters, in order to perform a pair-wise equivalence check, as per traditional metamorphic testing, but there is no reason to enforce that requirement, in case that there might be some checks of interest to be performed over a singular output. Any number of checks can be defined here, and they will all be emitted to the test file in turn after each metamorphic variant has been completely generated.

An example can be seen in Listing 3.1. We have defined two checks. The first check calls the internal `is_equal` function, which checks some internal notion of equality; it might be that the two tests have the same elements, or they have the same constraints. Based on the definition of the MRs and the fuzzer, we must know if it is appropriate to use this particular `is_equal` function here. This is an example of requiring knowledge of the library we are testing in order to write effective and correct specifications. The second check tests whether the cardinality of the two sets match. The MRs defined should emit equivalent sets, meaning that the cardinality should be maintained, thus making this an effective check. We also note that it might be the case that either `is_equal` or `count` might be buggy. Thus, it might be useful to have multiple, potentially redundant checks, in order to ensure such bugs get caught. Of course these functions might be used both within MRs and the fuzzer, if they are appropriately included in the specification.

The other two namespaces, namely `relations` and `generators`, are both comprised of further sub-namespaces, each one representing the notion of a high-level operation. Functions within each sub-namespace will each represent one alternative implementation of that particular operation, and each sub-namespace will have a homogenised signature (i.e., parameter count and types, and return type) shared by all MRs declared within. The major distinction between the two top-level namespaces are the imposed type restrictions. While there is no type restriction over `generators` (or second-class operations), we enforce that `relations` (or first-class operations) both have at least one parameter of the type of interest (namely the same type as of the fuzzed input variables and of the metamorphic variants; further discussed in Section 3.4.4), and that the return type is the same. The naming convention is an artefact from a previous iteration of the tool, where `generators` would be 0-ary MRs used to create objects with interesting properties (such as an empty set, or the universe set, in the domain of mathematical sets).

An example of possible MRs for our set example can be seen in Listing 3.2. The first thing to note is this `placeholder` function declared in each of the operation namespaces. This is the way by which we make use of recursion in `MF++`. This is an expanding marker, which `MF++` will know to replace with a randomly selected implementation of the respective operation. Further details about the recursion mechanism and implementation are available in Section 3.4.2. These `placeholder` functions can then be used at any point where there is a call to an operation which is abstracted as a high-level operation. For example, in the `de_morgan` implementation of *union* (line 12), we replace every instance of an operation with that of a high-level operation. There are two benefits to this. First, the trace of using this particular implementation becomes more

Listing 3.2: Example of MRs for the set library

```
1  namespace generators {
2    namespace empty {
3      set placeholder(set);
4      set base(set s) { return empty(); }
5      set empty_sub(set s) { return intersect(s, placeholder); }
6    };
7  }
8  namespace relations {
9    namespace union {
10     set placeholder(set, set);
11     set base(set s1, set s2) {  return union(s1, s2); }
12     set de_morgan(set s1, set s2) {
13       s1 = complement::placeholder(s1);
14       s2 = complement::placeholder(s2);
15       set s1_int_s2 = intersect::placeholder(s1, s2);
16       return complement::placeholder(s1_int_s2);
17     }
18   }
19 }
```

complex, thus potentially exercising the SUT to a higher degree. Second, we could lift this particular implementation of *union* to a generic set domain specification. We can then provide base implementations of these high-level operations making use of functions from a variety of libraries implementing set arithmetic. This allows us to reuse these metamorphic specifications across libraries with somewhat minimal effort; in our current iteration of `MF++`, we split the metamorphic specification into domain-related MRs, and library-related MRs — all the domain-related MRs make exclusive use of high-level operations, while library-related MRs only provide base implementations of high-level operations, and perhaps a handful of library-specific MRs. A final restriction for each operation is that it must have a least a *base* implementation defined, namely an implementation where there is no further recursion (e.g., lines 4 and 11). This is in order to ensure there is a choice to be made if the particular operation was chosen to be emitted at the maximum depth during the generation process.

Template File

In order to link all this together, the initial input to `MF++` is comprised of a `C++` file we consider a template file. All the expansions and syntactic markers are added here, and it is this file which, after `MF++` actions are applied to it, becomes the eventual test case. Declaring the specifications in header files and including those files inside the template file makes them available to the Clang parser, which means the respective nodes will be contained within the produced Clang AST to be used by `MF++`. This includes headers containing the metamorphic specification, adapted SUT headers with exposed functions and types for fuzzing, and headers containing

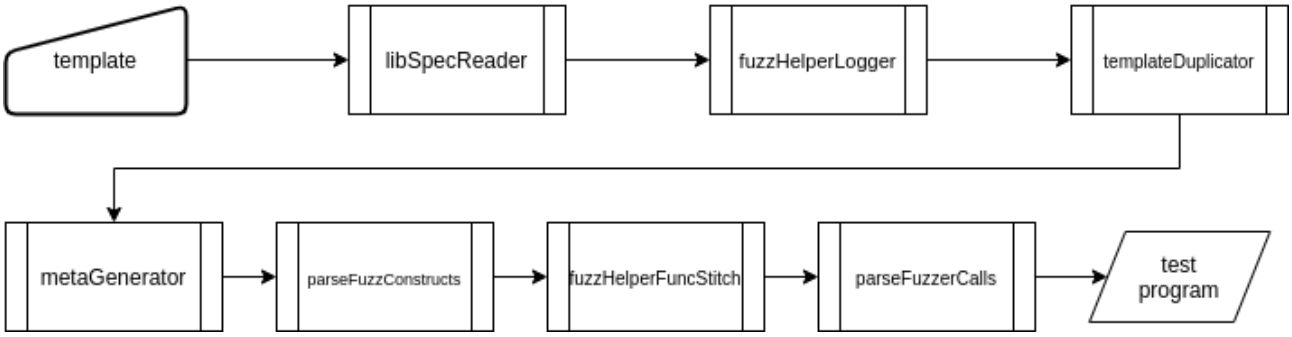


Figure 3.4: Internal pipeline of `MF++` consisting of Clang Actions

any further fuzzer helper functions.

However, the purpose of the template file is beyond just simply being a vehicle for all this information. It is also used to hold any SUT-specific setup that might be needed, and is beyond what the fuzzer prepares. For example, it might be the case that a particular SUT might require a state be initialised, or certain flags be set, or even some cleanup that might be needed at the end of the test. These can be added to the template file; `MF++` will not affect any code that is not explicitly marked for its use via syntactic markers. Additionally, due to how the fuzzer is built to make use of existing in-scope variables declared before the fuzzer call, it means specific variables can be prepared for use. These might be singleton variables, that must be used to create objects, and are vital for fuzzing, or they might be objects with very specific properties, that we might want to set and not randomly create. For example, if our example set library might have the notion of n-dimensional sets, and a `space` type object containing the number of dimensions be required to initialise one, we might want to have a single instance of such a `space` object, built with a fuzzed number of dimensions (via `fuzz_rand`). Then, constructors for such n-dimensional objects that require a `space` object as input will have our prepared instance for use, ensuring all n-dimensional sets reside in the same space.

3.4.2 Using Clang Libtooling

The high-level pipeline of how `MF++` operates can be seen in Figure 3.4. We distinguish seven actions, each of them performing a specific action in order to transform the given "template" (further described in Section 3.3.1) into an executable file calling the API of a library under test, integrating fuzzing and metamorphic testing. We now delve into more details regarding the purposes of these actions, and some implementation particularities, in order. We also distinguish transformative and non-transformative actions, where the former emits a transformed version of its input, which then must be reparsed by the subsequent action.

In addition to the internal implementation of metamorphic testing with high-level operations,

it is also important to consider the interface between the user and MF++. This includes how the user is expected to present the required user-provided data, such as the MRs and the data to be used by the fuzzer. This information definition already poses a rather high overhead, therefore it is important to try to design this interface in a way that is familiar and straightforward for future potential users. After a few iterations and brainstorming sessions, it was decided that the most natural way of asking users to write the required information would be through native C++ source code. As C++ is such a complex language, the only choice would be to make use of existing parsing utilities contained within the LLVM suite, more specifically, the Clang LibTooling library. This would provide access to the entire suite of utilities provide by the LLVM project, particularly the parser and subsequent Clang AST, which should cover all required use cases for our purpose.

Internally, we initialise a `clang::tooling::ClangTool` object, which defines the main entry point of our tool. Then, we define a series of `clang::ASTFrontendAction`s, with an associated `clang::ASTConsumer`. Each invocation of an `ASTFrontendAction` will internally perform the parsing process, and allow us to work at the level of the Clang AST, then delegate the actual work process to the associated `ASTConsumer`, where the work is being done. Our consumers are primarily implemented around Clang's matcher[3] and visitor[4] objects. The reason for using these objects is that they can identify parts of the AST of interest, which can then be used to emit the code we require in our eventual test case. Therefore, what MF++ does via its various actions and consumers is to read in Clang AST data, then emit an appropriately transformed version of this data, according to the process described in Section 3.3.3. We note here that it is important to make the distinction that the AST is not transformed, but rather that we use it to read in the data of interest, which is then used to construct the source code of the eventual test case via a `clang::Rewriter` object. The main reason for this is that the AST itself is immutable. What this means is that as we perform a transformation that a future action might depend on, we must emit the transformed program, then re-parse it from scratch.

`libSpecReader` and `fuzzHelperLogger`

These two actions are both non-transformative and serve to parse information contained within the template file and to store it in memory for use by further actions. `libSpecReader` logs information related to the API of the library under test. This means function signatures and object types to be used by the fuzzer. The main vehicle by which this information is

[3] https://clang.llvm.org/docs/LibASTMatchers.html
[4] https://clang.llvm.org/docs/RAVFrontendAction.html

presented is via an *annotation* attribute with a specific string which is hard-coded into `MF++`. The full annotation is `__attribute__((annotate("expose")))`. A proper attribute could have been implemented, but that would require modifying the Clang source code, and was deemed unnecessary for the current requirements of the tool. A clash is unlikely in any case, even if the annotation attribute allows for any user-provided string to be passed as a parameter. The implementation involves matching AST nodes of specified types (such as `CXXMethodDecl` or `TypeAliasDecl`) which implement types and functions of interest, and storing the required information into memory.

Similarly, `fuzzHelperLogger` is used to find additional functions that might be provided by the user in order to more coarsely control the fuzzing process. We call these *fuzzing helper functions*. For example, the user might wish to ensure that all sets have the same number of constraints, but the API of the SUT only provides a function which creates a set from an arbitrary number of constraints. It would be possible to not expose the API constructor, but to provide a helper constructor function which ensures the number of constraints is constant. Similarly, it is possible to enforce constraints on certain operations (division is a specific example) to avoid undefined or unsupported behaviour. Finally, multiple operations could be implemented in a single helper function, but done so in a restricted manner, as some of the operations might have unwanted side-effects when used more generally, but produce a known result if used in a controlled fashion. The logging is again done via matching, ensuring that these helper functions are placed in a specified namespace (in the current iteration, that being `fuzz::lib_helper_funcs`). Subsequently, these functions will be used by the fuzzer in the way the annotated library functions are used, with no distinction.

`templateDuplicator`

The first transformative action is used to prepare the eventual input variables. In the template file, the user can provide a sequence of instructions which represent the process by which the input variables are produced. However, we must ensure that there are sufficient input variables available for when the metamorphic variants are generated. Thus, we chose to duplicate this sequence of operations as many times as the specified number of input variables (which is a runtime parameter passed to `MF++` rather than something computed from the list of MRs; the tool will complain if there are insufficient inputs at generation time, however). In order to ensure correctness after duplication, we modify the names of all variables in a manner akin to single static assignment [4, 64]. Each duplicated block of instructions has an index which

is appended to each defined variable, and each instance of a read from that defined variable. The process involves visiting all nodes within the sequence of instructions for specifying inputs, and identifying first `clang::VarDecl` nodes, which declare variables, then corresponding `clang::DeclRefExpr` nodes, which reference other objects. If a `DeclRefExpr`, which references a variable we have identified, was declared within the input specification sequence, then we log it. Additionally, we log all the `clang::Stmt` within the specification sequence. Finally, we use a `Rewriter` to write the sequence of statements in sequence, duplicated as many times as required, but identify `clang::SourceLocations` at the end of variable declarations (`VarDecl`s) and respective uses (`DeclRefExpr`s), and append the current duplication index. We also use the `Rewriter` to remove to initial input specification sequence. Finally, we emit the rewritten code to disk, to be parsed by the following action.

We note here how all input variables at this point might seem completely identical, as what this pass does is duplicate code and ensure variables are distinguished across duplicate code blocks. We note that the template file allows for special syntax to be called which will be expanded into a fuzzed sequence of API instructions in the `parseFuzzConstructs` action. While there is still a semblance of similarity, this is intentional, as we thought it would be good to have some level of control over the properties of the input variables. Similar to fuzzing helper functions, the user might ensure the input specification ensures a specific property is held by the input variables, and then write the MRs assuming that property holds.

`metaGenerator`

The next transformative action has to do with generating metamorphic variants. First, we note that as the prior action was also transformative, at this point we must reparse the code. In this particular situation, the prior action duplicated the fuzzing specification to be used to produce our concrete input variables to be used by the metamorphic tests. We must reuse these variables, and it makes more sense to read them from the Clang AST (which comes with additional information that might prove useful). This can lead to future extensions, where input variables are identified based on certain properties (which means it might be not necessary that variables marked as input variables are used exclusively). Currently, it is the case that we identify input variables based on a reserved name. We must also note that the reparsing step means that references to the prior AST produced in previous actions are now invalid. This is why we must ensure that any information we would like to maintain between transformative actions are kept in our own data structures.

Going back to the `metaGenerator` action, it expands a known function call in the template file into the metamorphic variants, as discussed at length in Section 3.3.3. The first step is actually logging MRs defined by the user. We will distinguish three sorts of MR categories: (a) `checks` comprises of the user provided checks to be conducted in pair-wise fashion over the metamorphic variants, (b) `generators` are equivalent to second-class MRs, and (c) `relations` represent first-class MRs. We expect another level of indirection within `generators` and `relations`, representing the particular high-level operation the MRs implement, but otherwise, each category is comprised of a set of functions. We log all this information in memory, in a series of maps.

The first step in the actual generation process is to randomly produce the sequence of operations. This is done by selecting a permutation of all available `relations` operations, and this will be the basis of expansion into the actual metamorphic test. We iterate over the sequence, and select an appropriate implementation. The only limitation here has to do with the depth, as we would like to avoid theoretical infinite recursion (remember that an MR implementation could call a high-level operation, which will recurse into another implementation, and so on). Thus, we distinguish *base* MRs, which will not contain additional recursion, and can be safely emitted once the depth limit has been reached.

Once an implementation has been selected, we must emit it in our test file. This is done by constructing the instruction string by string. There are two main considerations to be made: how do we handle metamorphic variant declaration and updating, and how are the implementations emitted (especially considering recursive implementations). The former issue is handled rather easily, as either the implementations will return an object representing the result of applying the operation, or the changes will be made in-place. This is based on the SUT implementation itself, and the distinction at the level of `MF++` is made based on whether the operation declarations have return types or not. If they do, then we generate an appropriate name for the metamorphic variant (predetermined within `MF++`), into which we will store the result of calling the MR implementation. If we are processing the first operation in the sequence, we additionally emit the type of the metamorphic variants before the name of the variable, in order to declare it. We also distinguish metamorphic variant variables by an index.

For reference, an example code snippet is given in Listing 3.3, which implements the top-most example given in Figure 3.3. Here, our base metamorphic variant name is simply `r`, and the `_1` is used to distinguish if from other variants. Observe that the first operation in the sequence (line 8), a *union* operation implemented as calling the `base_union` implementation, also serves as declaring our `r_1` variable.

The other consideration has to do with how we emit calls the chosen implementations. Recall

Listing 3.3: Example of one generated metamorphic variant

```
1  set union_1_0(t1, t2) {
2    t1 = intersect_with_universe(t1);
3    base_union(t1, t2);
4  }
5
6  int main() {
7    [..]
8    set r_1 = base_union(i1, i2);
9    r_1 = r_1;
10   r_1 = union_1_0(r_1, i3);
11   is_equal(r_1, r_0);
12   [..]
13 }
```

that the implementations are provided as functions by the user. For base implementations, with no recursion, we can simply emit a call to the respective function (lines 3 and 8), and ensure the header where the particular function is defined is included during the testing process. However, when recursion is involved, it means that an operation call must be replaced with a call to another chosen implementation, meaning that we can't simply call the function definition. The chosen solution is to transplant the entire function definition (lines 1 to 4), append indices to ensure there are no name clashes, and update the body of the function appropriately, replacing recursion call points (line 2). Then we can emit a call to this particular function implementation. As we are working at the level of the Clang AST, this is not a particularly hard task theoretically. We identify the `clang::FunctionDecl` representing the particular implementation of interest, and obtain the corresponding `clang::SourceRange`, then copy that code over before the `main` declaration in our test file. We also update the name of the function by inserting text to the end of the appropriate `clang::SourceLocation` of the `clang::FunctionDecl`, and we can construct the name of the function to be called. Finally, we can identify points of recursion due to the design of the user-provided specification, as they will be implemented as calls to a function named `placeholder` (more details are provided in Section 3.4.1). Similar to above, we can identify the `clang::DeclRefExpr` node corresponding to the recursion point, and replace it with the another similarly specialised function implementation. Regarding parameters to the new function calls, they are passed as parameters to the `placeholder` call in case of recursive calls, or are inserted by name in the case of implementing a sequence operation.

Finally, we can emit all required checks (line 11), with appropriate parameters (again, that would be the current metamorphic variant, and a reference variant, usually variant with index 0). These function are emitted similarly to base implementations, as there is nothing inherently unique between calls beyond the parameters.

`parseFuzzConstructs`

At this point, we have almost everything put in position. One aspect we did not talk about yet is how the fuzzer is called. Similarly to the aforementioned usage of a function call `placeholder` to represent a recursion point within an MR, we have a function call to be expanded into the result of a fuzzer invocation. These fuzzer calls can either be placed within MRs, or they can be duplicated during the `templateDuplicator` action. The purpose of the `parseFuzzConstructs` action is to first identify these fuzzer expansion points, to pass the required information to the fuzzer to perform the fuzzing process, then replace each point with the respective output from the fuzzer. Due to the design of our fuzzer implementation, this operation is rather self-contained, and the only requirement to pass to the fuzzer itself is the user-defined data described in Section 3.3.2, and the type to be produced. Another consequence of the fact that each fuzzer call is completely disjoint from the rest is that there is no reuse of objects between fuzzer invocations. However, we note that other objects in scope before the call to the fuzzer invocation are available to be used during the fuzzing process. Internally, this is achieved by identifying the node where the fuzzer expansion is present, identifying the top-level node containing this expansion call (which would either be the `main` function for calling the fuzzer for input variables, or the function representing a specialised MR implementation if the fuzzer is called as part of an MR implementation), iterating over each `clang::Stmt` to identify `clang::VarDecl`s available to be used in the fuzzer (and additionally the `clang::ParmVarDecl`s representing parameters passed to the specialised MR implementation), then replacing the fuzzer expansion invocation with the fuzzer output.

We note here that the fuzzer implementation itself is part of a previous iteration of `MF++`, and has not yet been migrated to make use of Clang libtooling due to time constraints. A custom interface has been engineered between the current `MF++` implementation and the fuzzing implementation of the prior iteration. This means that at any point we can just link the interface to a libtooling implementation of our fuzzer, once one is available, with minimal to no changes of the current code.

`fuzzHelperFuncStitch`

The `fuzzHelperFuncStitch` action is similar in nature to how recursive MRs are implemented, but at the level of the fuzzer. We remind the reader that the user of the tool can provide their own functions for fuzzing, beyond those available only in the SUT API. We call these functions fuzzing helper function. In addition, we also allow the user to use the fuzzing capabilities built-in

MF++ within the definition of these functions. As such, similar to recursive MR implementations, each call to a fuzzer helper function must be distinct, as the body might be unique across calls due to the fuzzer calls. Thus, we must create a unique instance for each fuzzer helper function call. However, unlike the way we handled recursive MR calls, where we create a new separate function definition for an implementation, then just call that function, here we transplant the body of the function at the appropriate location.

For an example, let's assume we have the following helper function declaration:

Listing 3.4: Example of a fuzzing helper function

```
1 set get_set_with_even_elems() {
2   set s = new_set();
3   int count = fuzz::fuzz_rand<int>(0, 10);
4   for (int i = 0; i <= count * 2; ++i) {
5     s.append(rand()); }
6   return s;
7 }
```

The function calls the fuzzer (line 3) to get a random number between 0 and 10, inclusive. As this function might be called multiple times, and each time we might want to fuzz a different number, we create a separate body for each call, in order to have distinct `fuzz_rand` calls (the reason for this will be discussed in the next action description). Then, if we assume that we have a line that says `set r  get_set_with_even_elems=`, this will be expanded by the `fuzzHelperFuncStitch` action to:

Listing 3.5: Possible expansion of a fuzzing helper function

```
1 set s_5 = new_set();
2 int count_5 = fuzz::fuzz_rand<int>(0,10);
3 for (int i_5 = 0; i_5 <= count_5 * 2; ++i_5) {
4     s_5.append(rand()); }
5 set r = s_5;
```

We should note two things in the expansion. First, we must ensure there is no name clashing happening, which means we must ensure variables local to the helper function are made unique. This is done by, again, appending a global index to the variable names. Also note how the return instruction in the original function declaration has now been changed to just write the value to the variable to which the call was assigned to.

`parseFuzzerCalls`

The final action is another expansion transformation. In Listing 3.4, we have made use of a particular function call: `fuzz::fuzz_rand<int>(0,10)`. This represents a direct call to the fuzzer, and should be replaced with the appropriate output at the source level. As the fuzzer has internal capabilities of producing random values of select primitive types, it might be useful to ensure that all randomness goes through the fuzzer. Currently, fuzzing certain primitive types via `fuzz_rand`, and fuzzing objects of known types via `fuzz_new` (which is handled by the `parseFuzzConstructs` action) are the only two methods of interacting with the fuzzer at the template file level, but the infrastructure is there to implement more potential fuzzer calls. The main reason behind preferring this approach than to calling the in-built `C++` random utilities is to have an exact idea at compilation time of what the program does, without needing to actual run it (as all `C++` random decisions are made at runtime).

3.4.3 Implementation Choices

This section contains some general information about design choices made regarding some aspects of implementing metamorphic testing with high-level operations. Some of these refer directly to general aspects of the approach, which we have implemented for our specific implementation.

Ensuring Fuzzing Termination

There is always a random possibility that a fuzzer generates terminal objects, without the need to recurse further. However, it might be the case that this leads to intractably large programs to be generated. To ensure a reasonable generation for our tests, both in terms of generation time and test-case size, we impose a depth-limit on the fuzzing process. Whenever the fuzzer chooses to recursively produce a function invocation when it needs to generate an object, we increment the depth by one. Once the maximum depth-limit has been reached, we impose that the fuzzer use a terminal input (i.e., any object of the corresponding type in scope, or a constructor function). This requires that for each API type exposed to the fuzzer, there is a method of generating that type without further recursion. There is a small exception, in the sense that we include the option of calling defined functions for which parameters already exist in scope, either by declaration within the template file, or as a consequence of prior generation.

Controlling MR Recursion

Similarly to our fuzzing implementation, this recursion could in theory be infinite, and the main way we prevent that is by imposing a maximum depth that can be recursed towards, with each concretization of an operation along the chain incrementing that depth. Once the depth is reached, we enforce that a concretization without further recursion is chosen, similar to terminal inputs int he case of the fuzzer. This adds another restriction upon the MR specification, that each operation must have at least one concrete implementation which does not make use of any recursion. We have another parameter by which we control the MR recursion, which we name *pruning*. While not exactly true to its name, it involves adding an in-built weight to non-recursive MRs as we advance in depth in the generation process, relative to the maximum depth. We distinguish three types of pruning that `MF++` supports:

none There is no artificial weighting done.

linear The chance of choosing a terminal input increases linearly, with the chance of choosing a non-recursive MR concretization increasing by the same amount each step. In our implementation, each time we choose the concretization, we draw a random number between one and the maximum depth value. If that number is lower than the current depth, then we choose a non-recursive MR concretization.

logarithm We choose a non-recursive MR concretization if $\log(depth + 1)/\log(max\_depth + 1) > rand(0, 1)$. This strategy skews the process by choosing to emit non-recursive MR concretizations much earlier than the other strategies, and was implemented as a more aggressive pruning choice, making longer sequences less likely than the previous two options. This is useful for libraries for which the number of operations affect runtime in a non-linear fashion.

The choice of a specific pruning algorithm, in our experience, is determined empirically. The primary motivation was to add a bit more direct control (beyond random choice of concretization) over MR recursion (as discussed in Section 3.4.2), in cases where libraries would present many timeouts on generated test cases. We still wanted to generate *some* longer recursion sequences, but not so many that we would generate tests that are likely to timeout. We note that the choice of pruning strategy is another test generation parameter, similar to the sequence length, or number of produce metamorphic variants.

Second-class MRs

We mentioned that the primary property of second-class MRs is that they do not have the type constraints placed upon first-class MRs due to the interaction between input variables, the sequence of high-level operations, and the computation of metamorphic variants. More than that, due to the fact that second-class MRs need not carry any information, they can be used to create objects of interest from scratch. But the more interesting use-case is that we can focus on declaring MRs for other types than the selected type of interest. For instance, in our running set example, we could have a second-class operation be something like *set_int_to_set*, which takes as input a `set_int` and returns a `set`. With this, we could then define operations that work on `set_int` objects, which, due to MR recursion, could also in turn create a rich subsequence of operations as part of the sub-tree of a sequence-level operation. While there will be no explicit mixing between two sub-trees of types contained within second-class operations, due to the sub-trees across sequence-level operations being completely disjoint, we expect, implicitly, that the use of these second-class operations is somehow encoded implicitly. Alternatively, another use case could be to create objects with particular values of interest. In the domain of mathematical sets, we can make use of universe and empty sets to define additional MRs (e.g., $S = S \cap \mathbb{U}$).

Random parameter choices

While we simplified the discussion of how other variants are generated in the practical example, there is one consideration to be noted. The choice of parameters must be consistent across variants. That is, when a parameter for the first variant is chosen to be concretized to a specific input (e.g., the first operation in our example was concretized to an invocation which took i_1 and i_2 as parameters), it must be the case that the corresponding operation across all the other variants will take the same inputs as parameters (e.g., we can see in Figure 3.2 that all the i parameters for each operation are the same, and the v parameters are in the same position, but have different indices — this is due to the fact that they correspond to the currently computed value of the respective metamorphic variant). Due to this, and to simplify the generation process, we further enforce that there is a common signature for an operation, and that all MRs will respect that signature. This is to ensure that everything is in lockstep. We have not encountered a situation where this limitation did not allow us to define an MR of interest, and the simplification helps better debug the specifications, therefore it is useful to have practically.

Listing 3.6: Common header file to define MF++ syntax sugar

```
1  namespace fuzz
2  {
3      void start() {}
4      void end()   {}
5
6      void mfr_fuzz_start(std::string s) {};
7      void mfr_fuzz_end(std::string s) {};
8
9      template<class T> T fuzz_new();
10     template<typename T, typename U> T fuzz_rand(U min, U max);
11
12     void meta_test() {}
13 }
```

Syntax sugar

We mentioned that MF++ is able to parse certain syntax and expand it during the generation process. In order to comply with the fact that the file to be parsed by Clang must be valid, we define a generic header file which contains definitions of these syntactic markers, along with an empty-body declaration. As the file is rather short, it is reproduced in its entirety in Listing 3.6.

We distinguish five types of syntactic markers, two of them coming in pairs, which we briefly describe. The first pair of markers, `start` and `end`, are used to mark the start, and respectively, end, of the block of code that represents input variable initialisation code, and should be duplicated by `templateDuplicator` (Section 3.4.2). They should be written by the user as part of the template file, and they will be deleted during the expansion process. The next pair of markers, `mfr_fuzz_start` and `mfr_fuzz_end`, similarly distinguishes a block of code which was generated due to a fuzzer call. All code within the two markers is fuzzed, and its main purpose is to forward to information to the reducer (as part of the Fuzzing Reduction opportunity detailed as part of Section 5.2). It is automatically generated by the fuzzer. The next two markers, `fuzz_new` and `fuzz_rand`, are markers written by the user which should be expanded into fuzzer calls at generation time. The former will produce an object of class `T`, where `T` should be a valid type made known to the fuzzer, and the call will be replaced by a sequence of instructions to generate a random object of type `T` (as described in Section 3.3.2). The latter marker generates a single object of a primitive type, and the parameters are limits of that type. For example, if type `T` is `int`, then the parameters would represent the lower and upper ranges, inclusive, that the fuzzed value will be within. Finally, `meta_test` represents the expansion to the metamorphic tests, including all variants with their respective concretized sequences, and the checks for them.

3.4.4 Caveats and Limitations

In this section, we shall discuss some known limitations and caveats related to the current implementation of `MF++` and the design surrounding it. These have mostly become apparent during our intensive usage of the tool, and have been discussed at fair length. Even with these limitations, we still believe the tool is fit for purpose, and effective at producing tests that find bugs and exercise the SUT.

Need for Manual Input

In order to use `MF++`, there is a fairly high initial overhead of understanding how to write the specifications for the future SUT, finding MRs themselves for the SUT, and any potential debugging that might need to be done over the specifications. Furthermore, at the point where one might consider investing all this effort, there is no obvious gain: the effectiveness of applying the tool to the respective SUT will only become apparent once tests have been produced and pushed through (which is of course directly correlated to the quality of the specifications provided). It might be more worth it for library developers to focus on writing high-quality and highly specialised tests for their applications than to spend time preparing these specifications for unknown gain.

To counter this point, we argue that, while there is no initial way to quantify the utility of using `MF++`, it at least provides a practically infinite set of tests to be executed by the SUT, making the initial overhead pale in comparison to practical gain. Additionally, since the quality of the tests is directly proportional to the expressiveness of the specifications, small incremental updates to the specification might trigger large gains in the efficacy of the tests. An exercise in potential efficacy of `MF++` is whether a specification can be produced to generate an existing test case of interest, which can be a starting point into writing a strong specification.

Another argument against manual input is the potential of automating the written specifications. One point of discussion would be the current state of the art in MR generation techniques using machine learning, which, to our knowledge, exist [39, 40], but the quality and utility of found MRs is very volatile. There should be some oversight assessing the efficacy of these automatically generated MRs. However, an automated method of gathering these MRs would complement the specification writing aspect of `MF++` nicely, but is orthogonal to the tool itself. But the major point against automatically generated MRs is the general applicability. We believe `MF++` can be used for any library, as long as MRs exist to be discovered for that library. But further than that, each library is unique, and the best way of using `MF++` is leveraging that

uniqueness to produce specialised MRs, highly specific to the particular SUT. And while of course an automated technique might stumble upon such MRs, we believe the experience of library developers will make them obvious quite quickly as they get used to the specifications of `MF++`.

Top-level Single-type Focus

We note that there is a restriction placed on the signatures of functions concretizing the same operation, and this is primarily exposed in the concretization of the sequence of operations. We must feed the result of one MR application to the subsequent MR application in order to not discard results of previous computations. What this means is that we must constrain all MRs to have at least one parameter of a type of interest, as well as return an object of the same type of interest. Further, due to how we use our input variables, it is the case that at least one of these input variables must be of the type of interest (depending on how many parameters of that type are required, as per the above not about input variables). This might seems like a very strict limitation, affecting both the space in which we can fuzz (as we will heavily focus on fuzzing objects of the type of interest), as well as the kinds of MRs we can write, since one parameter and the return type are predefined. However, we almost entirely circumvent this issue by distinguishing *first-class* and *second-class* MRs. In this system, we shall impose these type limitations only on first-class MRs, while allowing second-class to be defined without restrictions. This distinction is made more obvious by introducing *recursive* MRs in our MR definition.

For the homogeneous type requirement across variants, we believe that this is alleviated by the distinction between first-class and second-class operations. As second-class operations have no restrictions, that means any sequence of calls can be produced, intermingling types, at one level of depth. This would make it indistinguishable from having a top-level sequence of operations that is composed of multiple types: a specification could present MRs in such a way that one single operation could be theoretically expanded in multiple operations across different types, due to recursion. A very simple example would be to just define a top-level operation which takes a parameter of the homogeneous type, but then uses the parameter to generate an object of another type of interest, via second-class operations, which are then used to operate on the original object. More specifically, we could "unfold" all the recursion in the chosen MRs to have a pseudo-sequence of high-level operations per variant.

The homogenised signature across operation implementations is not strictly required, as

it can be the case that the signature will have a set of all required parameters, and only a selection is used in one particular implementation. The return value should be consistent for an operation, however. If there are equivalent types, then multiple operations could be defined for these types, or alternative types could be implemented with second-class operations, and a conversion operation be defined. The biggest reason for the homogenised signature is that it makes it much easier to handle the `placeholder` function inside `MF++`.

Discussion on Sequences of High-level Operations

There are a number of questions that might arise in regards to our design of generating these sequences of high-level operations, which lie at the foundation of our implementation of metamorphic testing. One direct question might be how can we assess whether this design is helpful in finding bugs. Well, a naive answer is that it has been able to find bugs in practice (Section 4.4.1). Other than that, it's hard to perform a quantitative analysis on its overall bug-finding ability. We aim to generate large sequences, in order to explore a larger space of the underlying SUT, to more likely trigger bugs (discussed in Section 3.3 of the `Csmith` paper[80]). One other aspect that makes it very difficult to quantify the effectiveness of the testing is the presence of the user-provided ingredients. As they greatly affect the tests produced, it is impossible to know what a perfectly defined set of specifications would look like, in order to be able to assess whether such a specification is guaranteed to find known bugs under our method. For instance, if the specification does not cover certain parts of the SUT, it is impossible for it to find any bugs in those uncovered regions.

Furthermore, we do not impose any restrictions in the way high-level operations and sequences interact with one another, including no set dependencies. The distinction between first and second-class MRs is mostly for ease of defining the user-data, and ensuring that the sequence of operations is as simple and dependency-free as possible. One can argue that the true testing is done starting from one recursion level, where *all* provided MRs are free to be generated. However, the final metamorphic variant should contain all effects of API calls along the generation path. Therefore, a buggy API call should cascade towards a faulty metamorphic variant, leading to a failing check.

One other question is whether it is the interaction between MRs, namely the sequence of high-level operations, and recursion, which is the main cause of bugs, or a single MR being able to expose a particular defect of some API calls. This is again part of the design of the user-provided specification. We chose to keep our MRs simple, and allow the recursion to generate

richer SUT calls. We did find bugs that required few SUT calls, which could have been part of a single MR, but were separated in our design, and the random aspect of our approach eventually generated the correct sequence. This could be an indicator to further refine the specifications, by including such problematic sequences directly as first or second-class MRs, without relying on randomness to generate them.

Implementation of `fuzzHelperFuncStitch`

This particular action was implemented during the translation of the previous iteration of `MF++` into the current variation, using libtooling. As such, one particular design choice was included, which was the fact that the test case was going to be a single straight-line `main` function, rather than multiple functions with calls inserted where appropriate. This was due to the limitation of how code was generated in the previous iteration of `MF++`. At that point, we were more concerned with ensuring the code was logically sound, more so than with the design of the tool. This was due to the translation process being a very laborious undertaking. Thinking back, it would be better to not transpose the bodies of the helper functions, and just emit new function declarations for each use, similar to the MR recursion process.

However, one main limitation still remains. Consider lines 3 to 4 of Listing 3.5, reproduced here:

Listing 3.7: Limitation of helper fuzzer functions

```
1 for (int i_5 = 0; i_5 <= count_5 * 2; ++i_5) {
2     s_5.append(rand()); }
```

Note that there is a `rand()` call inside the `for` loop. That is because the fuzzer would replace a call to itself with the produced outcome at the source code level, which means it will be there at compile time. There is currently no implementation of having the fuzzer produce different outcomes during runtime, as it would be required by having a fuzzer call within a loop. The workaround is, as shown in the example, to use the in-built `C++` randomness capabilities.

However, there is a reason we might want to have all random choices controlled through our tool. That is that we might want to experience with different random generation engines, or we might want to use the random generation log internally for some reason. There are tools which can perform reduction at the level of random choices (such as Hypothesis [50]), meaning we'd get a free implementation of a reduction approach for `MF++`. There are further arguments to be made against this very fine control over randomness, such as if the seed in the test-case is an outcome of our fuzzer, then reverting the fuzzing choices would affect the choices made

by the native C++ random engine, or that such a fine level of control is unneeded, as reduction can be performed in other ways, and as long as we ensure the test is reproducible, that should be sufficient control of the random choices made.

3.5 Related Work

In this section, we shall provide some details of other testing techniques that we are aware of, which are similar in nature to your proposed approach of metamorphic testing with high-level operations, or tools similar with MF++, which fit the same category of synthesising useful test cases for a SUT.

3.5.1 Property-based Testing

Property-based testing (PBT) is a testing technique that is similar in functionality to our proposed approach of metamorphic testing with high-level operations. A definite definition of PBT is difficult to come by[5], but rather the technique is more defined in practical terms. The first instance of property-based testing was QuickCheck [23], a tool to generate automatic test cases in Haskell, and it seems to have been the original tool to spawn the field of property-based testing. The main features of the tool focus around defining properties (which is akin to the operations we define in our approach), and randomly generating inputs to check the properties against, via generators (which can be thought of as similar to how we randomly choose implementations for our sequence of operations to create metamorphic variants). Another similarity is how generators can be improve upon manually to increase the quality of the generated input, similarly to how a better quality specification leads to better tests being generated by MF++.

While there are surface level similarities between PBT (or at least QuickCheck-like PBT), we believe there are fundamental differences between the two approaches. First of all, fuzzing, while a core component of MF++, is not necessary for our metamorphic testing with high-level operations approach. For us, it was more convenient to have a never-ending supply of input data at demand, and customisable for any SUT we desire, and while it is true that the fuzzing component was able to find bugs, the core of the technique is still the equivalence check done at the level of metamorphic variants. Although there are PBT tools, such as SmallCheck [67], which perform exhaustive search over inputs, in favour of using random generation. This is still means that the generation process is built within the technique itself, rather than it being

[5] https://hypothesis.works/articles/what-is-property-based-testing/, accessed 24th of August, 2021

unconstrained.

We can perform comparison between the implementation of fuzzing withing `MF++` and that of QuickCheck, or the notion of using generators to guide the random generation. In our tool, fuzzing is done by referring to the API of the SUT allowed by the user, which means that an expert of the SUT can quickly understand what kind of objects the fuzzer would produce. Conversely, defining a generator in a PBT tool would mean interacting with the API of the tool as an interface to the API of the SUT. While this level of indirection might mean a PBT generator could easily be reused for multiple SUTs, as the input would be library-agnostic, it is opposite with our intent of having a very SUT-directed approach to testing.

To conclude, practically the two techniques can do similar things, and with sufficient work, it might be possible to implement both the kind of tests that `MF++` produces in a PBT tool, and the PBT workflow in `MF++` for a specific SUT. However, the methods by which they interact with the SUTs are different, with `MF++` allowing for a closer interaction with the source code of the SUT, and PBT abstracting away the more technical details.

3.5.2 Automatic Test-case Generators

At the end of the day, `MF++` is a tool which generates tests for a given SUT automatically, while having a required initial manual effort cost in the form of defining the SUT specifications. There already exist tools and techniques which can automatically synthesise tests, with little to no manual effort required.

One such example is **EvoSuite** [5], a framework which is able to automatically synthesise tests fulfilling a given coverage criterion, implementing *search-based testing*. It is automated in the sense that there is an in-built algorithm, and a fitness criterion guiding the search space, and only a test suite is required as input in order to make EvoSuite work. Of course, the criterion can be adjusted as per the needs of the SUT. EvoSuite uses mutation algorithms in order to alter the entire test suite closer towards the chosen criteria. Thus, it can produce tests which complement the existing test suite of SUTs. However, these tests might also require manual inspection, to ensure that they are valid; it might be the case that a test produced by EvoSuite uses some undesired implementation of the SUT in order to achieve it's coverage. One major drawback of EvoSuite compared to our approach is that it does not have an in-built oracle; indeed, the paper mentions that is beyond the scope of the work. Additionally, supposing an existing test case is integrated in the test suite, it is not future-proof; future changes might break assumptions made by an EvoSuite-produced test, which might be counter-productive as

an SUT evolves. As MF++ is more suited to be used consistently alongside the SUT, it should evolve along with it, and the core specification is unlikely to change.

3.6 Summary and Future Directions

In this chapter, we have presented our take on metamorphic testing, introducing **metamorphic testing with high-level operations**. We have discussed the main theoretical points of the approach, and have presented MF++, an implementation targeting C++ software libraries, built around metamorphic testing with high-level operations, but also implementing other features (such as fuzzing, and recursive MRs) in order to improve the testing process. The tool is aimed at software developers, and while it requires a fairly high initial manual effort to start, it offers access to a practically infinite supply of tests to exercise the SUT, with maintenance and further improvements requiring little further effort.

Future Work There is still a lot of space to improve MF++. First, the development of the tool is highly coupled with the types of libraries that we have tested, which almost exclusively reside in more mathematical domains. Trying to push through libraries from additional domains might expose certain limitations that we are not aware of in the implementation of MF++. This would also include the Clang AST parsing, as we do not exhaustively handle all nodes types in MF++. Additionally, even for the libraries that we are testing, we can always further improve the specifications, in order to exercise parts of these libraries that are unavailable to MF++ with the current iterations of the specifications.

There are also some orthogonal directions we can take our work in. For example, as we have domain-specific specifications, which can be concretized into applications of a specific SUT, we could potentially create a SUT-agnostic test for a given domain, and simply link the base function calls against calls in various SUTs. This would open the door to **differential testing**, allowing us to execute one test over a variety of SUTs, and to observe any inconsistencies that might lead to potential bugs.

A natural follow-up for MF++ would be to find actual developers willing to put in the time to integrate their SUTs with our tool. The interface between the human and MF++ has not been thoroughly tested, and has been done on a best-effort basis. However, the tool is meant to be used by these developers, making an evaluation of this interface a top priority for the future usability of the tool.

4 Case Studies of Metamorphic Testing for C++ Software Libraries

In this chapter, we discuss in detail how we applied `MF++` over a variety of libraries in the domains of Presburger arithmetic and satisfiability modulo theories (SMT). While we discuss a total of 6 SUTs over which we have applied `MF++`, we will be more verbose when discussing how we integrated `Z3` and `isl` in our tool, as they were the first SUTs tested in their respective domains. This includes discussing design choices in the specification of `MF++`, as well as observations and particularities of the two APIs, and how these are handled in `MF++`. To evaluate `MF++`, we discuss a number of bugs found during our application of the tool over the 6 SUTs, as well as results of a preliminary experiment to evaluate the use of `MF++` as the first part of a workflow focused on enhancing test suite coverage.

4.1 Motivation

The most effective way of assessing the efficacy of the approach discussed in Chapter 3 is to practically apply it to existing, real world software. Fortunately, there are numerous real-world open-source projects available for this purpose. These libraries, with in-development versions available on demand, provide a great source of potential test material for `MF++`, while also providing us with feedback from the actual developers regarding the utility of the tests we generate that mark potential inconsistencies in the SUTs.

In addition to gauging the bug-finding capabilities of `MF++`, we are also interested in evaluating the simplicity of the interface between a potential SUT and `MF++`. As developers of `MF++`, we are already greatly familiar with this interface, thus any shortcomings obvious to us would be multiplied manyfold to a third-party developer. Having experience with a wide variety of distinct libraries, and getting an idea of potential requirements and ease of specification writing for these libraries is essential in order to polish the design of the specification writing, trying to lower the burden of writing these as much as possible.

At the same time, writing effective specifications requires a level of expertise for the desired libraries, due to the chosen design of `MF++`. Thus, we must strike a balance when choosing suitable libraries for us to test, as we must understand the API and know how to use it correctly, as well as be aware of any potential pitfalls present in the API that might be obvious to an expert user. Thus, we firstly focused on libraries for which we had practical experience working with, in order to minimise the need of learning the underlying SUT, then branch out to similar, but completely new, libraries in the same domain as the initial test batch.

We note that the, while the chosen libraries in this chapter are mathematical libraries, this is just a taste of what kind of MRs can be given to `MF++`. We can ask whether MRs for use with `MF++` can be derived for an arbitrary library in an arbitrary domain. The answer is that this is SUT-specific: we first need to investigate a concrete SUT to be able to say whether it is amenable to testing with `MF++`. An example would be, in the domain of filesystem operations, applying `move` to some source and destination inputs would be equivalent to a `copy` from the same source and destination, followed by a `remove` of the original source. However, if the library were to not have a `remove` operation (say, for security purposes, users are provided with a stripped library which does not allow for outright removal of files), then the previous MR would not be possible in that instance. But similarly, we might have some obscure functionality outside the scope of usual filesystem libraries, such as an `encrypt`/`decrypt` pair, which would allow for a larger space to define MRs. To conclude, being able to define MRs in the scope of `MF++` is dependent on the SUT for which those MRs are to be used for testing.

4.2 SMT Solving Libraries

The implementation of `MF++` was done in parallel with developing specifications for SUTs appropriate for our needs. These SUTs were chosen due to a mix of our familiarity with them, as well as suitability of applying metamorphic testing. The first set of such SUTs belong to the domain of Satisfiability Modulo Theories [8] (SMT) solvers.

SMT solvers are generally used in one specific manner: a first-order logic formula is produced, optionally containing a set of free variables, which encodes some question of interest, then the formula is attempted to be solved, with reference to some underlying knowledge, known as the *theory*. There are various ways of encoding a formula, but SMT-LIB2 [7] represents a unified standard that defines a basis for implementing theories and logics. Due to the general availability of SMT-LIB, and the fact that all our chosen libraries can be referenced to the SMT-LIB standard, we expect some commonality between the APIs of these SUTs.

We first describe how SMT solvers work at a high-level. SMT formulas are, as their name implies, SAT formulas [35] with some additional domain knowledge, represented by the theory. As the SAT problem is a known NP-complete problem, there is no guarantee that a solution can be found for any arbitrary formula within a reasonable timeframe. Furthermore, the theories supported by SMT solvers can be **undecidable**, for example when quantifiers or arbitrary integer arithmetic is included. An example of a theory is the theory of Ints[1], which defines the `Int` sort, as well as operations including addition, multiplication, or comparison, similar to arithmetic operations over mathematical integers. However, note that these theories might have specific distinctions made. For instance, the theory of Ints defines `mod` and `div` according to Boute's Euclidian definition [11].

In addition to the overall theory, SMTLIB also distinguishes *logics*[2], which further specify the properties of the formula to be solved. There are a large number of such defined logics, allowing for specific optimisations to be made within such a logic. For instance, the logic **QF_NIA** represents the quantifier-free non-linear Integer arithmetic logic, while **QF_LIA** represents the quantifier-free linear Integer arithmetic logic. These logics specify what kind of operations are expected within the provided formulas to be solved. Therefore, a formula attempted to be solved in a **QF_LIA** logic should not expect any existential or universal quantifiers, nor should it expect multiplication of free variables. Allowing for these assumptions means that the solver could employ more specialised techniques to attempt and solve the given problem.

One major difference between the chosen tools is the kind of theories the tool implements. For example, as we will see, `Z3` [25] is more inclusive, having implementations for roughly all SMT-LIB defined theories[3]. Conversely, `Boolector` [12] only supports the logics `QF_ABV`, `QF_AUFBV`, `QF_BV` and `QF_UFBV`, from the BitVector theory.

We note that these SUTs have been approached by us in a grey-box fashion. We studied the API, and used our domain knowledge, to have sufficient information regarding a reasonable base type to target our testing, and what kind of operations would be generally used over that type. We have tried to deepen our understanding of certain libraries, particularly `Z3`, in order to evaluate how a more in-depth specification would affect the quality of the generated tests. However, we most likely have not come close to what an expert in the library might write, nor have we sought out to write specifications targeting critical areas of the code, or understand specific peculiarities of respective libraries. Even with this, we were able to make good use of

[1] `http://smtlib.cs.uiowa.edu/theories-Ints.shtml`, accessed 30th of August 2021
[2] `http://smtlib.cs.uiowa.edu/logics.shtml`, accessed 30th of August 2021
[3] An exhaustive list of which can be found at `http://smtlib.cs.uiowa.edu/theories.shtml`, accessed 26th August 2021

the exposed APIs, and write tests effective at finding bugs in most of these libraries.

4.2.1 SMT Solver APIs

The exposed API of these SMT solvers have a few similar components. They offer types representing a **variable**, with an associated **sort** if the respective solver supports multiple theories, a **formula**, constructed via variables, support **operations** (again, as determined by the logics and theories implemented), and there is a mechanism by which to **solve** a formula. We shall briefly discuss particulars of these elements for each of the four SMT solvers we test.

Z3

Z3 [25] was the first SMT library chosen to apply MF++ to, mainly due to our familiarity with working with it in prior projects, as well as being a well-regarded and long-standing project. The main advantages of Z3 in the context of MF++ are that it implements a variety of theories, has a rich and fairly complete C++ API, and has additional non-standard supported operations in addition to those defined by SMT-LIB, which means we can express more interesting MRs using Z3. As a project, Z3 is well known in the SMT community as one of the major available solvers, and is well supported on GitHub. Considering its longevity, any bugs we would be able to find in the logic of the solver would indicate that our approach is effective.

A formula in Z3 is expressed as a `z3::expr`. This includes everything, including variables, free variables, constants, and entire formulas (comprised of various operators, variables, and terminals). The API is meant to use `z3::expr`s, alongside a plethora of functions, which internally perform the correct operation based on the sort of the given inputs. Therefore, at the C++ API level, we need not worry whether we are working with integers or bit-vectors, as everything is a `z3::expr`. However, using an unsupported operation (such as `abs` over a bit-vector variable) will lead to an exception being triggered. The other interesting components from the Z3 C++ API are a `z3::context`, which all `z3::expr`s must reference, and which holds information about the formula, such as the logic used, and parameters set, and a `z3::solver`, which is the main interface to calling the internal solver. Note that Z3 performs **no** computation of interest before the solver itself is called, and it is mostly internally building the AST of the formulas created.

In terms of MF++, abstracting sorts is slightly problematic. As the entire semantics of the variables in the program is contained within the type of a variable, that means mixing sorts would be impossible, as the tool cannot distinguish between an integer sort `z3::expr` and a

boolean sort `z3::expr`. In order to overcome this issue, in a fashion similar to how helper fuzzer functions are defined (Section 3.4.2), the user can use `typedef` to introduce their own types in the specification. As such, for `Z3`, we define two additional types within the `fuzz` namespace: `bool_term`, which will represented boolean sort `z3::expr`s, and `int_term`, for integer sort `z3::expr`s. In the code itself, both are `typedef`'ed to the base `z3::expr` type. However, then we can write the MRs and checks in the specification referring to these types, ensuring that the correct variables are passed throughout the generation chain. Another benefit of this approach is the fact that more complex types can be abstracted to simple specification-defined types. Again, the tool only parses those Clang AST nodes which we are aware of, and does not do so exhaustively. Furthermore, the Clang AST might be changed at any point, which would require internal modifications to `MF++`. Thus, simplifying a potentially complex AST representation of some type to a single `typedef`, both enhances the compatibility of `MF++` with generic `C++` code, and leads to easier to read tests being generated.

While there is a fair amount of interacting with the `Z3 C++` API during the fuzzing of `z3::expr`s and the generation of metamorphic variants, the interesting functionality of `Z3` is not exercised until we actually attempt to solve a formula. This is done via *checks*, and it is arguably important to get these right, and to ensure they exercise a sufficient amount of the underlying code. At the moment that the check is generated, what we expect to be available to us is two metamorphic variants (a reference variant and the currently generated variant), which are expected to be **equivalent** by construction. This, again, is due to the choice of operations, and the fact that all implementations of an operation should yield the same result. It would be fine to implement an operation which just maintains the satisfiability of a given input, without maintaining any other notion of equivalence. The first check to be performed should thus ensure this equality holds. Thus, for two given inputs, i_1, i_2, we can check equality by checking that $i_1 \neq i_2$ is **not satisfiable**. Note that, due to the incompleteness of SMT solvers, we do not check for **unsatisfiabilty**, as another possible result is **unknown**, to indicate that the solver is unable to come up with a conclusive result. In more detail, for all the free-variables contained within both i_1 and i_2, there is no assignment for which $i_1 \neq i_2$ holds. The possible values for these free variables will be determined by the theory of the formulas.

We mentioned free variables as part of the expressions we generate. This is the main driver allowing us to produce interesting formulas. However, the main issue with free variables is that we must very carefully control their use in order to ensure the expected equality property holds across metamorphic variants, and that we do not overload the solver by having too many such free variables (while it is hard to define the expected performance of these solvers, as

the underlying problem is often undecidable, depending on the chosen theory, simply adding more free variables leads to an exponential increase in complexity). Thus, we do not allow the fuzzer to generate these free variables on demand. In the template file, we define a number of free variables, which are then made available to use as a terminal value to the fuzzing process. Thus, every variant will use free variables drawn from this limited pool. Furthermore, there are only two instances of fuzzer calls where these free variables might be used. First is during the generation of input variables. This maintains consistency of use to free variables and the inputs are used consistently for all variants. Second, we allow a second-class operation of fuzzing random expressions. While this means that a variant that might contain this second-class operation in its recursive generation tree could access free variables that are not in use by other variants, potentially causing a divergence, we very carefully control where this fuzzing calls are called from to ensure this does not happen. For example, we might desire a randomly generated expression be created, which we then use to subtract from or divide by itself.

We might want to exercise additional features in the checks we implement. For instance, a *model* is an assignment of all free variables within a formula which makes it satisfiable, and is a basic feature of all SMT solvers (as, of course, a satisfiable formula must have a corresponding model that achieves that satisfiability). The above check does not exercise any capabilities related to models. Another check we implement (Listing 4.1) creates two formulas, $i_1 == 0$ and $i_2 == 0$ (we note that 0 is an arbitrarily chosen value, as the purpose of the check is to check the model capabilities of the SUT), then checks the satisfiability of both. Due to the two input formulas being equivalent, we expect the satisfiabilities of these equalities to match, and check as such (lines 25 to 28). We note that the expected satisfiabilities are `SAT`, `UNSAT`, or `UNKNOWN`. The latter is usually returned if the formula becomes too hard for the solver, and it decides to give up. We treat this result as acceptable, to account for potentially using undecidable theories in our tests (which we do, as will be discussed in Section 4.2.2). Furthermore, as the set of free variables used within both formulas are the same, as discussed above, we know that if one of the two formulas is satisfiable, then a model for it should also satisfy the other formula (and vice-versa) (lines 29 to 33).

One small issue with this check is how `Z3` produces a model for a formula. As mentioned before, we provide a pool of free variables to be used by the formulas at any point in their construction, and that a formula might use additional free variables, which will be either part of dead code or simplified away to other known values. As such, semantically, all metamorphic variants will make semantic use of the same subset of the larger given pool of free variables. However, it seems that when evaluating a formula with an existing model, it must be the

Listing 4.1: Metamorphic check exercising model features of Z3

```
void
check_exprs_are_zero(z3::context& c, fuzz::FreeVars& fvs,
      fuzz::int_expr e1, fuzz::int_expr e2)
  z3::solver solver(c);
  solver.push();
  solver.add(z3::operator==(e1, 0));
  z3::check_result result_1 = solver.check();
  assert(c.check_error() == Z3_OK);
  z3::model mdl_1(c);
  if (result_1 == z3::sat)
  {
      mdl_1 = solver.get_model();
      for (fuzz::int_expr e : fvs.vars)
      {
          z3::func_decl cnst_decl = e.decl();
          if (!mdl_1.has_interp(cnst_decl))
          {
              z3::expr zero_val = c.int_val(0);
              mdl_1.add_const_interp(cnst_decl, zero_val);
          }
      }
  }
  solver.pop();
  [..] // Get result_2 and mdl_2
  if (result_1 == z3::sat)
  {
      assert(result_2 != z3::unsat);
  }
  if (result_1 == z3::sat && result_2 == z3::sat)
  {
      assert(mdl_1.eval(z3::operator==(e2, 0)).bool_value() == Z3_L_TRUE);
      assert(mdl_2.eval(z3::operator==(e1, 0)).bool_value() == Z3_L_TRUE);
  }
  assert(c.check_error() == Z3_OK);
}
```

case that the model itself is complete (i.e., it contains assignment to all free variables used in that formula). This means that we must "pad" our models to ensure that there is at least an assignment for all potentially used free variables (lines 13 to 20). As the assignment does not matter, just that it exists, we simply assign each free variable the value zero.

CVC4

Another well-known SMT solver, and with a development age comparable to Z3, CVC4 [9] also has support for multiple theories and a few non-standard operations, providing a robust build infrastructure and seemingly rich system and regression tests. As its features are comparable to Z3, CVC4 was chosen as a pair library to test with Z3: we can validate that a potential bug that we discover in one library is not present in the other, thus increasing our certainty that it is a legitimate bug, and not that we are using the respective library in an undefined way. We note that while initial testing involved CVC4, the project was later succeeded by CVC5. The general principles are applicable to both versions of the tool, but in the interest of consistency, we will be talking about CVC4 generally, unless it is to do exclusively with CVC5.

Similar to Z3's z3::expr type, CVC4 provides a CVC4::Term type. The usages are similar: it is a type holding an abstract SMT formula, but it does come with an inner sort handled appropriately in the API, and checked on function invocation. As opposed to Z3, which implements a variety of operations over z3::exprs, CVC4 provides functions to create n-ary operations via overloaded mkTerm functions, with varying numbers of input terms, from 0 to 3, and including a vector of inputs. The choice of operation created by mkTerm is controlled by an enum parameter passed to it. In the context of MF++, this makes it rather hard to encode. As MF++ works primarily at the level of function invocations, this particular implementation detail would require the fuzzer to be aware of one of the parameters passed to the function being generated. The initial design of MF++ did not consider such a situation. However, this can be worked around, by defining wrapper functions with predetermined enum values already in place. Thus, supposing there are two CVC4::Term variables in scope, named t_1 and t_2, if we would like to generate the addition of the two, we would normally call the function mkTerm(CVC4::PLUS, t_1, t_2). Rather than somehow constrain the fuzzer to correctly emit CVC4::PLUS as the first parameter, which would be akin to presenting some semantics to the fuzzing process, we can define a function which takes two CVC4::Term parameters, and passes them on to a mkTerm call with the first parameter set to CVC4::PLUS, returning the value of that function invocation. We can similarly define wrapper functions for other operations of interest. This does mean more

initial work to write these wrapper functions, and the inability to directly expose the library data for the fuzzer in the library source code, but is arguably a more elegant solution than mixing semantics and syntax in the fuzzer.

One other distinction to Z3 is that there is no direct mechanism by which to obtain and reuse a model. We would be able to print a model out after executing a call to the solver, and then parse the output appropriately, and emulate the existence of a model by enforcing values on required variables. However, from an engineering perspective, this would prove time-consuming for little gain, and could potentially be emulated by a test which has those values already set, instead of allowing them to be unbound.

Yices2

Yices2 [29] is an SMT solver working over integers, reals, bitvectors, and booleans, and covering a rather wide selection of related theories. It provides the option of using alternative solvers than the built-in one in the backend, while supporting both its own input language, as well as SMT-LIB2. Importantly, it provides a C API (compatible with C++ files by the nature of the two languages), and is able to handle SMT_QF_NIA with further dependencies.

The major difference between Yices2 and the previous APIs is the fact that Yices2 only provides a C API. Luckily, this is compatible straight out of the box with how MF++ interprets specifications. We do still have to emit C++ files, as the fuzzing and metamorphic generation functionality within MF++ is written in C++ format. However, this has no bearing on interacting with the Yices2 C API itself.

In terms of internal functionality, the core SMT-LIB2 implementation is what we expect for operations. One particular detail of Yices2 is that it seems to simplify the internal representation at every step it can, rather than when explicitly called, as in Z3 and CVC4. Coming at the cost of up-front performance, this approach would circumvent both expensive simplification calls over particularly complex expressions, as well as potential bugs arising from attempting to internally handle such complex expressions in a solver call.

One infrastructure difficulty is the need to get additional dependencies in order to make use of the non-linear capabilities of Yices2. What this means further is that tests that we generate are not reproducible without these capabilities enabled, and would not be thus generally applicable if we emit expressions that make use of non-linear algebra. This is more a tool consideration than an issue with MF++, however, and of course we could ensure the specification does not generate non-linear expressions.

`Boolector`

The final SUT to discuss is `Boolector` [12]. Similar to `Yices2`, it provides a `C` API, and not a `C++` one. As noted, `Boolector` only provides support for certain bit-vector logics. Given this, there are numerous bit-vector specific operations implemented in `Boolector`. However, the overall approach does not change much compared to previous specifications, especially `Yices2`. The APIs of the two SUTs are nearly identical, with a large overlap in both operations and chosen API function names for those operations (with appropriate library name as an identifier as the main method of distinguishing between API provenance). The similarity of the two APIs is another positive point in favour of testing the utility and design of `MF++` specifications, and the slight SUT-specific differences further enabled the checking of SUT-specific MRs implemented in our design of specifications.

4.2.2 Choosing Theories and Logics

Having included `Z3` and `CVC4` in our list of tested SMT solvers, this gives us full reign to choose any theory we would like, as they both support a large variety of both theories and corresponding logics. Starting by selecting a logic is important in the context of `MF++`, as it shapes restrictions that might need to be placed on the fuzzer, and what kind of operations should be exposed to it. Integer theory seemed to be a good candidate, as it would offer us access to a variety of MRs by using known arithmetic identities, beyond those of Boolean theory, and it would also be distinct enough from the domain of Presburger arithmetic. The chain of SMTLIB2 logics defined over the Integer theory define, in order of complexity, `QF_IDL`, `QF_LIA`, and `QF_NIA`, as one possible branch of logics we can choose. We have not considered `QF_IDL`, due to our limited experience with Integer Difference Logic [41]. We thus chose to test both `QF_NIA` and `QF_LIA`, due to their similarity at the API level, meaning in terms of that operations they expose. In order to make our specifications compatible with `QF_LIA`, we would use the sledgehammer approach, and disable any exposed operation or MR that would make use or lead to non-linear results.

After some practical experimentation with both theories, we observed that linear tests would finish quite quickly. This led us to think that maybe these tests were too simple, and would not exercise the solver sufficiently. In contrast, there would be a large number of execution timeouts for non-linear tests, even when the limit would more than double (from a default of 120 second execution timeout to 300 seconds). With this data, we decided to pursue testing the `QF_NIA` logic, but attempt to rein in the execution time.

In our first attempt, we attempted to limit the number of non-linear objects that would be produced, by directly removing MRs from the specification. We were unable to find a good balance between having interesting MRs which did add non-linear arithmetic, and an acceptable timeout ration (at this stage, we would still observe a timeout rate of over 50% for generated tests). The second idea was to try and impose a limit on the depth of the generated test cases. This ended up becoming the *pruning* feature of the generator (discussed in Section 3.3.3). With logarithmic pruning, we were able to produce tests which had some interesting sequences, but removed enough of the rest of the test such that it would execute sufficiently quicker. We thus settled on testing `QF_NIA` as one of the main theories for SMT solvers.

When we added `Boolector` to our list of tested SMT solvers, we obviously needed to expand the scope of the tested theories to include a bit-vector logic. A good starting point was `QF_BV`, the quantifier-free logic for bit-vectors, allowing us to make use of all bit-vector features within SMTLIB, but restricting the use of quantifiers. Similar to our approach to choosing `QF_NIA`, this was a suitable decision to make, to assess the initial utility of `MF++` for a new tool.

4.2.3 Unifying SMT Specification

After some practical experience with writing specifications for these four libraries, as well as understanding their underlying APIs, we can observe a number of core similarities.

They have a common type for a formula A formula seems to be a basic construct for each of the four libraries. Concretely, they are represented as `z3::expr`, `CVC4::Term`, `term_t` in `Yices2`, and `BoolectorNode`. Furthermore, all libraries internally contain the notion of a sort, and each instance of a formula has such an internal categorisation, which is concretized once operations are applied over these instances.

They implement the SMT-LIB2 standard By having a unified standard, this means we can categorise the features of each solver, and there is a baseline of expected implemented operations for a given theory (although it might be the case that a library can implement supplemental operations). Another consequence is that libraries can consume SMT2 format input, allowing for crosschecking. We note that `Yices2` and `Boolector` do offer their own formats, which would require additional overhead of understanding it to query internal status.

They have similar critical internal structure Beyond the similarities at the level of formula implementation, there are a number of core constructs shared across all libraries. We

Listing 4.2: Abstract specification example

```
namespace metalib {
namespace rels {
namespace neg {
  int_term placeholder(context, int_term);
  int_term neg_by_sub(context c, int_term t) {
    return rels::sub::placeholder(c, t, rels::add:placeholder(c, t, t)); }
}
namespace sub {
  int_term placeholder(context, int_term, int_term);
  int_term sub_by_add(context c, int_term t1, int_term t2) {
    return rels::add::placeholder(c, t1, rels::neg:placeholder(c, t2)); }
} // namespace sub
} // namespace rels
} // namespace metalib
```

Listing 4.3: Abstract specification concretized for `Z3`

```
typedef int_term z3::expr;
typedef context z3::context;
namespace metalib {
namespace rels {
namespace negation {
  int_term base(context c, int_term t) {
    return z3::operator-(t); }
}
namespace subtraction {
  int_term base(context c, int_term t1, int_term t2) {
    return z3::operator-(t1, t2); }
} // namespace sub
} // namespace rels
} // namespace metalib
```

distinguish the notion of an overarching **context** object (`z3::context`, `CVC4::Solver`, `context_t` in `Yices2`, `Btor` in `Boolector`) to which all formulas are referenced. This context can be used to interface with solving capabilities.

With these core elements, having selected a logic to test against (for example, `QF_NIA`), we can set high-level operations, which we know will be present, and write our MRs in the `MF++` specifications exclusively referencing these high-level operations. We must also define an abstract "formula" type which we must write these MRs over. Then, we can link such an abstract specification with a given SMT solver by providing base concrete implementations of these high level operations which implement the API of the chosen SUT, as well as using `typedef` to link the abstract formula type with the concrete type of the SUT.

Inspecting the theory of Ints[4], there are a number of operations defined (e.g., negation, subtraction, multiplication), which can serve as our high-level operations. These can be defined

[4] `http://smtlib.cs.uiowa.edu/theories-Ints.shtml`, accessed 12th of October 2021

Operation	Implementation

negative	`-s1`
	`0 - s1`
	`s1 - 2 * s1`

modulo	`ite(s2 != 0, mod(s1, s2), s1)`
	`ite(s2 != 0, abs(rem(s1, s2)), s1)`
	`ite(s2 != 0, s1 - (s1 / s2 * s2), s1)`

identity	`s1`
	`s1 * 1`
	`s1 / 1`
	`ite(s1 == abs(s1), abs(s1), s1)`
	`ite(fuzz_false(), fuzz_int(), s1)`

Table 4.1: Selection of SMT MRs used in MF++ testing

in a separate, SMT specification header file, alongside any interesting concrete implementations making use of all these high-level operations (Listing 4.2). Then, we can define a separate, SUT-specific header file, which links the abstract specification with the concrete API of the SUT (Listing 4.3). In the SUT-specific header file, we can also include any additional specific operations or implementations to be used in the generation process.

This allows us to reuse domain specifications across any number of SUTs, as well as providing a template for future additional SUTs that we might want to integrate with MF++. It also provides a clear distinction between what is pure MF++ specification, and what is required to interface with a new SUT, and provides an initial low-overhead method of quickly integrating MF++ with a new SUT, provided a domain specification is readily available. Finally, if the abstract specification itself is improved, then these changes should cascade to existing SUTs with no further changes, if no extra operations were added to the specification.

4.2.4 Example MRs

We illustrate a selection of MRs used for SMT testing in Table 4.1. Terms $(s1, s2)$ are inputs to the MRs, available either via fuzzing or as an intermediate MR variant generation result. Terms $(0, 1)$ represent the literal values 0 and 1, which within the actual MR specifications mean recursive calls to generator MRs for those values. We additionally note that any expression can be wrapped around a call to an `identity` operation (omitted in these examples for space). The remaining interesting constructs are `ite`, representing a ternary if-then-else function, and the two `fuzz_false()` and `fuzz_int()` calls, which are high-level operations fuzzing a known-to-be false boolean expression, or an integer respectively (via MF++'s syntax sugar mechanism,

as described in Section 3.4.3).

The first examples, of the `negative` operation, showcase how arithmetic identities can be used in the domain of `QF_NIA` SMT solving. This operation takes an expression as input and should return the negation of the expression as output. While the identities in this format do not appear particularly expressive, we note that due to recursive MRs, they can be expanded into very large sequences of API calls. Thus, even these simple MRs are enhance the testing process, while being easy to write and understand.

The `modulo` operation, as its name implies, takes two inputs and returns the value of the first input modulo the second input. We note the second implementation of this operation includes the `rem` function. We found this function while perusing the `Z3` documentation, and noticed that its functionality is equivalent to `mod`, but with different sign semantics. We noted this discrepancy can be avoided by wrapping the `rem` call in an `abs` call. We note that this result was due to practical experimentation with different variations of attempting to find equivalent uses of `mod` and `rem`. This was partly due to the fact that we were unable to find any documentation as to the exact functionality of the two functions. Nevertheless, `MF++` can validate its own specifications, by quickly identifying failing test cases. Furthermore, if there is suspicion of an MR being incorrect, then the specification can be minimised to include a very limited set of MRs, including that MR, with generated test cases quickly exposing the potential issue.

The third `modulo` implementation makes use of domain-specific knowledge due to the `Ints` theory we are testing. As the division used in the `Ints` theory is integer division, as per Boute's definition [11], then $s1/s2$ yields an integer quotient. Multiplying the result by $s2$ again, and subtracting from the original $s1$ should yield the remainder of dividing the two. The addition of this MR proved fruitful in uncovering bugs, and perhaps more such MRs using specific domain knowledge can be employed to further strengthen the testing process.

One other aspect of note is that we integrate semantics checks within the modulo implementations. As we can make use of direct API code, we use the inbuilt `Z3 ite` function to check if the divisor is 0, as modulo 0 is an invalid operation. Thus, pre-conditions required by certain operations can be directly included in the implementations of those functions, ensuring that no undefined behaviour or invalid operation is generated.

The `identity` operation further showcases some more instances of common arithmetic identities applicable to the SMT domain, exemplifying that initial testing can be performed even with simple MRs, which will be expanded appropriately. The final implementation shows how fuzzed objects can be used within MRs themselves. In this instance, we ensure the if-then-else

always follows the `false` branch, due to the condition being guaranteed false on generation, meaning that we can potentially insert any expression in the then branch. In this case, we fuzz another fresh expression, but it could be possible to include an operation which produces API traces making use of undefined behaviour. As the code is guaranteed to not be executed by construction, it might prove an interesting challenge to the in-built optimisation passes, potentially triggering some other bugs.

4.3 Presburger Arithmetic Libraries

The second domain of choice is that of Presburger arithmetic [72]. Presburger arithmetic is a first-order logic over natural numbers containing a limited set of operations (integer addition, logical negation, logical implication, and equality). The main strength of the logic is to eliminate free variables, which are known to be particularly difficult to solve.

For the purposes of `MF++`, we consider our chosen SUTs as set arithmetic libraries, where the basic building block is a **set** object. Further, we will expect the usual operations over sets, including **union**, **intersection**, and **subtraction**. We note that both SUTs contain more than just set objects (`isl` for example has the notion of a **map**, or a **union set**), but we shall focus exclusively on sets, as **(a)** we were advised by `isl` developers initially that sets are the basic building block for the other types (we did not seek to find out if this was the case for `Omega`), and **(b)** we expect that a specification focusing on another type would mostly mirror a set specification, thus a deeper testing into sets might expose more bugs of interest

4.3.1 `isl`

As the first library that was used to build `MF++` around, and with direct access to developers, `isl` [73] was the one of the main driving pillars behind `MF++` development. Similarly to `Z3`, we dedicated considerable effort in testing `isl`. The library implements Presburger arithmetic, and has a lot of very domain specific features, such as n dimensional sets, and being able to compute the convex hull of such a set. As testing was done primarily in close coordination with two `isl` developers, Tobias Grosser and Sven Verdoolaege, the aspects of `isl` that we focused testing on were mainly chosen at their recommendation. One particular operation, namely *coalescing* [74], was mentioned as being generally under-tested, and making a good candidate on which to emphasise `MF++` testing.

For `isl`, we were provided with a generated `C++` interface, accessing useful internal `C` func-

tions. We mainly use the `C++` interface, even if it means that a function not exposed by the interface cannot be visible to `MF++`, as it contains most core operations we require. Further, we shall refer to types and functions in this `C++` API.

We start our discussion with fuzzing input variables for the metamorphic variants. A set in `isl` is represented by the type `isl::set`. Constructing a set in `isl` can be done in three main ways:

- read the set in from a string;
- define some `isl::point`s in an `isl::space`, and find the `isl::set` defined by these points;
- manually create constraints via `isl::pw_aff` (piece-wise affine expressions) inequalities formed of `isl::val`s (values)

We initially started with the third approach, which we named `isl_cons`, for `isl` constraint. In order to generate a `isl::set`, we start from `isl::val`, representing integer values in an `isl`-friendly format, combine them into `isl::pw_aff` (formally, piecewise quasi-affine expressions, defined in Notation 4.12 of the `isl` tutorial [76]), then further combine these `isl::pw_aff` expressions into sets. This approach has been lifted from the `isl` manual [75], which provides it as an example on page 45. This approach allows us to exercise types `isl::val` and `isl::pw_aff` during fuzzing, in addition to our main focus on `isl::set`. We then augment these sets by intersecting multiple multiple such sets together.

However, a main disadvantage of this approach is the fact that choosing to intersect sets can quickly lead to the empty set being produced, if there are two non-overlapping subsets generated at any point. And considering that there is no semantic restriction over generation of `isl::val`s and `isl::pw_aff`s, this case is very likely to be triggered. Using union instead of intersect poses a similar problem: we might generate constraints which, together, cover the whole space, leading to constructing a universe set, as opposed to an empty set. Nevertheless, even with this slightly flawed approach, we were able to discover a number of internal crashes in `isl`.

The second approach we attempted to generating interesting inputs, named `isl_point` is the second option outlined above: randomly generate some `isl::point`s in a given space (by randomly generating `isl::point`s, we mean setting their coordinates across all dimensions of the space to some random value), converting such a point into a `isl::set` object, the uniting these sets together. This approach does not suffer from the issue discussed in the `isl_cons`

generation method. By uniting subsets, we can never reduce a final set to the empty set. Furthermore, generating a point that covers the entire space is impossible, as we do not allow infinity as values that can be assigned to values of dimensions. This ensures we can create well-defined, and likely disjoint sets as our inputs to our metamorphic variant generation.

In terms of functions available to apply during the generation process, we constrain them carefully, to ensure we generate valid objects along the way to our final set generation. Thus, we allow usual arithmetic operations over `isl::val` objects, we provide a wrapper function which can set a specific, valid dimension of a `isl::point` to a randomly generated `isl::val`, and we also allow the generation of intermediate `isl::set`s, over which we allow the use of usual set arithmetic operations. There are some operations that would be interesting to allow during the fuzzing process, but are unfeasible without further constraints. An example would be `isl::set::project_out`, which projects a set across the given dimensions, reducing the number of dimensions of a set. The difficulty here is that sets must reside in the same space in order to be used in further operations. There could be workarounds, such as lowering the set residing in the larger space to the smaller-dimension space, or somehow emulating both sets residing in the larger space, but this might have further implications on the fuzzing process, and would affect our assumptions that input variables to the metamorphic variant generation process are compatible out of the box. We can, however, explicitly insert such a `project_out` call across all variants, to exercise this feature of `isl`.

For the metamorphic variants, we provide MRs mainly inspired from first-order logic identities, translated to the domain of arithmetic sets. The most `isl`-specific MRs are identities making use of `isl::set::coalesce` and `isl::set::detect_equalities` functions. For the equivalence checks, we use the in-built function `isl::set::is_equal` to check for equality across metamorphic variants.

4.3.2 Omega

The `Omega` [71] library was chosen as a second implementation of Presburger arithmetic, in order to mirror `isl` and observe the efficacy of our Presburger arithmetic specifications. Initially started as the *Omega test*, which would only manipulate constraints over integer variables, it evolved into the current system for simplifying and verifying Presburger formulas. As such, it implements similar operations and features as `isl`, theoretically allowing for cross-checking of generated test cases across the two SUTs. We note that preliminary testing of `Omega`, which identified issues affecting around 70% of our tests, and a lack of support for the library led us

Operation	Implementation
`complement`	`complement(s1)` `subtract(universe(), s1)`

`intersect`	`intersect(s1, s2)`
	`intersect(s2, s1)`
	`complement(union(complement(s1), complement(s2)))`

`union_assoc`	`union(s1, union(s2, s3))` `union(union(s1, s2), s3)`

Table 4.2: Selection of Presburger arithmetic MRs used in `MF++` testing

to abandon in-depth testing of this library.

It provides the same constructs and features that we use for the `isl_cons` specification. We consider the core element of `Omega` the type `Omega::Relation`, which is similar to an `isl::set`. The main difference between `Omega` and all other SUTs is that there are specific types for specific operation applications. For example, if we would like to apply a logical and operation over an existing `Omega::Relation`, we can use `Omega::Relation::add_and()` to obtain an object of type `Omega::F_And`, representing a functional application of logical and over the input relation. Similarly, adding equality via `Omega::F_And::add_EQ()` yields a `Omega::EQ_handle` object. Thus, in order to prototype our specification over `Omega`, we initially used a wrapper function taking as input an integer value to create a hard-coded `Omega::Relation` with a concrete application of logical and arithmetic operations. Further, there was also no direct check for equality between `Omega::Relations`, from what we have observed. Thus, in order to simulate an equality check between two `Omega::Relation` objects, we made use of the function `Must_Be_Subset()`, which takes two input `Omega::Relations`, and returns a boolean value, which we assume represents whether the first parameter is a subset of the second, from inspecting the comments in the source code. It should follow that calling the `Must_Be_Subset` function over two parameters both ways is equivalent to checking that the two parameters are equal.

Other operations over `Omega::Relations` mirror those over `isl::sets`, and the MRs we specify are first-order logic identities over booleans, translated to the domain of arithmetic sets. This prototype was sufficient to evaluate both the ease with which we can specify a new SUT in an existing domain, as well as `Omega` itself as a SUT (as discussed in Section 4.4.1).

Library name	inputs	tests	test-size	test-depth	prune-depth
Z3	2	5	4	2	logarithm
CVC4	2	2	5	3	noprune
Boolector	2	3	4	5	logarithm
Yices2	2	5	4	5	logarithm
isl	3	7	5	4	logarithm

Table 4.3: Chosen parameters during evaluation for SUTs tested with MF++

4.3.3 Example MRs

Some select MRs applied to the domain of Presburger arithmetic used in MF++ are shown in Table 4.2. Similar to the SMT example (Section 4.2.4, $(s1, s2, s3)$ represent inputs to the MRs. The universe() function call is used to generate a set containing all elements of the domain.

We observe a one-to-one mapping with known arithmetic identities in the MRs applied to the Presburger arithmetic SUTs: union associativity is equivalent to associativity of addition, and intersect MRs represent the properties of commutativity, as well as a slightly modified instance of an application of a DeMorgan law. The second complement MR is rather more domain specific: the complement of a set can be obtained by subtracting the set from a universe set.

The main methods in which we found bugs in Presburger arithmetic SUTs were SUT-specific instructions. For isl, these mainly focused on isl::set::coalesce() and isl::set::detect_equalities(), which take a set as input and perform internal operations to optimise the internal representation of that set. These two represent SUT-specific instances of identity operation implementations. For Omega, a function which triggered an unexpected error was the Must_Be_Subset function, given two copies of the same Omega::Relation object. Clearly, we expect a set to be a subset of itself.

4.4 Experimental Results

In this section, we discuss our findings during the evaluation process of MF++. We investigate two aspects of the tests MF++ produces. First (Section 4.4.1), the capability of finding bugs, and whether these are bugs that matter. Secondly, we investigate the potential of using MF++ as a coverage-enhancing tool, and perform some pilot coverage experiments using tests produced (Section 4.4.2).

There are a number of parameters that can be set to adjust the generation process of MF++.

These are:

- `inputs`, the number of input variables to generate via fuzzing;
- `tests`, the number of metamorphic variants to generate;
- `test-size`, the size of the sequence of high-level operations across all metamorphic variants;
- `test-depth`, the recursion depth for one high-level operation in the sequence — once this depth is reached, concretizations which contain other high-level operations are disallowed from being generated;
- `prune-depth`, the algorithm to be used for pruning (Section 3.4.3)

These parameters have been determined for each library empirically, to balance the amount of timeouts with the amount of fully executed tests, while aiming to generate larger tests. Thus, the main goal of choosing parameters was to keep the timeout rate under a threshold. A more exhaustive experiment, with more qualifiers, could be used to potentially find better parameter values, but our main goal was to execute more tests over the SUTs themselves, than over the generator. The chosen values for each library can be seen in Table 4.3.

4.4.1 Bug Finding

The main purpose of `MF++` is to find bugs in the SUTs it targets. We define this process as a sequence of actions: **(a)** a test executed triggers a failure, **(b)** the failure is investigated and triaged into either false alarms or potential positives, **(c)** positive failures are manually reduced to a simple example and re-evaluated, **(d)** and finally, positive reduced tests are submitted as issues.

During this process, we highlight two important steps. First, triaging bugs is very important. As `MF++` lies with one foot in the domain of fuzz testing, we must ensure that the bugs we find are both valid (i.e., not due to an incorrect specification), and of relative importance [22]. Initially, as generally the generated test cases would be too complex to manually go through by hand, we must look at the cause of the failure, both in the SUT, and the trigger point in the test case itself. For example, `Z3` has operations which operate over objects of type `z3::expr`, but check that the internal sort of these objects are valid (e.g., `z3::expr::abs` is not applicable to `z3::expr`s of sort `bv`). This can be regarded as a specification issue, but it does require internal and specific knowledge of `Z3` itself; it is reasonable that another SMT

Bug count	Library	Description	Library Commit	Report URL
1	Omega	Internal assertion failure	b601950aae5376cf234e05e43bea28c3891cedd8	https://echo12.cs.utah.edu/dhuth/chill-dev/issues/57
2	Omega	Internal assertion failure	n/a	https://echo12.cs.utah.edu/dhuth/chill-dev/issues/58
3	Omega	Internal segmentation fault	n/a	https://echo12.cs.utah.edu/dhuth/chill-dev/issues/59
4	isl	Internal assertion failure	db1090fb3bf1b7385c29ab829979a9afd939da33	https://groups.google.com/g/isl-development/c/11by6uHRNVA
5	isl	Internal assertion failure	44534fc75cb4d67f6aa74ec201b0b0bbcfd7ae2a	https://groups.google.com/g/isl-development/c/iO1QFBGfdcE
6	isl	Internal assertion failure	38f1a592e80176863a68a2bf305f4dfd6e1999cb	https://groups.google.com/g/isl-development/c/JFll6ZM6za4
7	isl	Internal assertion failure	dc9b84f72f24cf1817e7d4e0fbc146bd204a8f68	https://groups.google.com/g/isl-development/c/BxicmfMDNdw
8	isl	Internal assertion failure	179e221bbd892f327cfb7ba3ed408ae695d604aa	https://groups.google.com/g/isl-development/c/qMyF1QBHl90
9	isl	Metamorphic check failure	179e221bbd892f327cfb7ba3ed408ae695d604aa	https://groups.google.com/g/isl-development/c/BjxxUFI410c
10	isl	Metamorphic check failure	8ef424541905b8ec5878538d9fc5b774c0c98cc7	https://groups.google.com/g/isl-development/c/gZ4fGOsGpCE
11	isl	Metamorphic check failure	379be8a469f526c4970e47a826eb8c6736053634	https://groups.google.com/g/isl-development/c/qsQDKnfuRdg
12	isl	Infrastructure bug	n/a	https://groups.google.com/g/isl-development/c/M24UHP9lfCI
13	isl	Internal assertion failure	n/a	https://groups.google.com/g/isl-development/c/A7vN0dY-gE0
14	isl	Metamorphic check failure	n/a	https://groups.google.com/g/isl-development/c/yT2BfQ0Wt4w
15	Z3	Metamorphic check failure	8566d88b992610060a6523f28272d3384a2f2471	https://github.com/Z3Prover/z3/issues/2096
16	Z3	Metamorphic check failure	b4ba44ce9d39b784a07d9b6a878193af03946219	https://github.com/Z3Prover/z3/issues/2238
17	Z3	Metamorphic check failure	d70b63c8acf816e57adce30242c0c8a81be515f0	https://github.com/Z3Prover/z3/issues/2604
18	Z3	Performance bug	d0d06c288a76236ea6fb32b3858e7414e7d5f4c5	https://github.com/Z3Prover/z3/issues/4775
19	Z3	Asan error	n/a	https://github.com/Z3Prover/z3/issues/5435
20	Yices2	API bug	7e3815d5bbf80fd39155ca39bda9ac18af02465d	https://github.com/SRI-CSL/yices2/issues/353
21	Yices2	Metamorphic check failure	12378df46b35fe6e1f5b370d2797e1f290887a7a	https://github.com/SRI-CSL/yices2/issues/360

Table 4.4: List of all bugs found and reported during MF++ evaluation

solver might implement an `abs` operation over bit-vectors. We note here that versions of the SUT must also be taken into consideration—it might be the case that the bug has been fixed in a more recent version than the one it has been observed in (and conversely, the bug might have been recently introduced). Second, submitting a bug report is not always straightforward. Some libraries might impose certain requirements on the bugs that one might want to report[5]. Developers or maintainers of other libraries might be more difficult to track down. One such example is `Omega`, which does offer a repository on GitHub[6], seemingly abandoned. We were able to contact recent maintainers after a suggestion of seeking out a colleague of the former developers.

Throughout the implementation of `MF++`, we have found a total of 21 bugs across 4 of the 6 SUTs considered: 5 bugs in `Z3`, 2 in `Yices2`, 11 in `isl`, and 3 in `Omega`. All of these were reported to their respective maintainers. All, except the 3 `Omega` bugs, were subsequently fixed. In the following, we shall go into more detail about the process of reporting bugs for each of the libraries, and discuss further details for a selection of more interesting bugs. A list of all the bugs found is presented in Table 4.4.

`Z3`

We were able to find and report 5 bugs for `Z3` by using `MF++` alongside an `SMT_QF_NIA` specification, augmented with boolean expressions. We distinguish three kinds of tests. One bug was identified due to compilation with AddressSanitizer (asan) [70]. One other bug we classify as a *decidability* bug after initial triaging. Finally, the remainder of the bugs are metamorphic bugs, exposing internal `Z3` logic errors.

Metamorphic bugs We found three instances of tests where two expressions which should be equivalent by construction were deemed not so by `Z3`. They are all rather simple to express, but we must understand that the internal workings of `Z3`, and SMT solvers in general, are very sensitive, and even seemingly obvious solutions actually require a fairly involved solving process.

For a first example, consider the code snippets in Figure 4.1, representing Bug 15. We observe that the code snippets are equivalent. The main difference lies in line 5. Variable x, due to line 4, must have value -2. This means that the two expressions are expected to be semantically equivalent. Furthermore, we can manually compute the value of y, that being 0. Thus, the

[5] `https://github.com/cvc5/cvc5/wiki/Fuzzing-cvc5#general-guidelines`, accessed 7th of October 2021
[6] `https://github.com/davewathaverford/the-omega-project/`, accessed 7th of October 2021

```
1 (set-logic QF_LIA)
2 (declare-const x Int)
3 (declare-const y Int)
4 (assert (= x -2))
5 (assert (= y (- -2 (div ( * -2 x)
     -2))))
6 (assert (not (= y 0)))
7 (check-sat)
```

```
1 (set-logic QF_LIA)
2 (declare-const x Int)
3 (declare-const y Int)
4 (assert (= x -2))
5 (assert (= y (- -2 (div ( * -2 -2)
     -2))))
6 (assert (not (= y 0)))
7 (check-sat)
```

Figure 4.1: Example of Z3 internal solver bug

```
1 z3::expr r0 = z3::ite(z3::operator!=(y, 0), z3::operator-(x, z3::operator*(
     z3::operator/(x, y)), y)), x);
2 z3::expr r1 = z3::ite(z3::operator!=(y, 0), z3::abs(z3::rem(x, y)), x);
```

Figure 4.2: Example of Z3 metamorphic check failure

check (line 6) should fail, and we expect the solver to return `unsat`. In actuality, Z3 evaluates the left snippet as `sat`. Investigating the commit[7] required to fix the issue, it seems to have been due to incorrectly propagating the constant value of a negative divisor variable; this gives a clear example of the complicated inner workings of these tools. We also observe that the bug was seemingly introduced during a previous fix, as per the commit message. Being able to catch this issue with MF++ is also an indicator of its utility as a regression test tool. In this particular case, we note that the bug was reproducible entirely in SMT2 format. Tests from MF++ are initially in C++, but providing a test case with the buggy behaviour in SMT2 format ensures that this is an internal solver bug, rather than an API issue somewhere up the chain. The full bug report is available online [8].

A snippet of a second bug is shown in Figure 4.2, representing Bug 16. Given some input expressions x and y, we define two expressions, `r0` and `r1`, which we expect to be equivalent by construction. They are both instances of a modulo operation. In the case of `r0`, we use the fact that, as we are in the `SMT_QF_NIA` theory, division is actually integer division, meaning that only the integer quotient is returned, and the remainder dismissed. Thus, $x\%y == x - x/y * y$ holds. For `r1`, the `rem` function, which, from testing, outputs the same absolute value as the modulo, but with a distinct sign based on the operands. As such, applying the `abs` operation over the output of a `rem` operation is expected to yield the same value as a modulo operation. This identity is implemented as an MR in our Z3 specification, and required a fair bit of work and investigation of the provided documentation to create. Checking that $r0 \neq r1$ with specific

[7] https://github.com/Z3Prover/z3/commit/49a51a075776cd37126bf868cd92184e484752a7, accessed at 11th of October 2021

[8] https://github.com/Z3Prover/z3/issues/2096, accessed 11th of October 2021

inputs[9] yields, `sat`, while the expected return value is `unsat`, due to the expected equivalence of the two expressions.

The final issue, Bug 17, involves two fuzzed expressions, x and y, creating two solver instances, and checking the satisfiability of $x \neq y$ in one, and $y \neq x$ in the other. Of course, the devil lies in the actual fuzzing process of the expressions. What we observed is one of the checks yielding `sat`, while the other was found to be `unsat`. Furthermore, on a `Z3` web interface[10], the result was `unknown` for both formulas. This is not particularly surprising, as we are in the `SMT_QF_NIA` theory, which is undecidable. It is reasonable for the solver to not be able to solve formulas gives. However, if the solver returns a particular result, as was the case in testing `Z3` directly, then that value should be trusted. The issue was reported[11] to the `Z3` developers and subsequently fixed.

Decidability bug We initially classified bug 17[12] as a *performance* bug, but the actual fix proved that this was more of an internal API bug. The reason for the initial classification was due to the fact that for the formula `(assert (<= 0 (^ 2 -1)))` would yield `unknown`. The formula is trivially satisfiable, due to constant propagation. However, as we are in the theory of **non**-linear algebra, the theory itself is **undecidable**. SMT solvers usually employ various tricks and heuristics in order to evaluate the satisfiability of a formula, but there is no expectation that they should be able to solve any formula provided to them. In this instance, our initial expectation was that perhaps the nature of our formula, in conjunction with the lack of proper documentation for the implementation of the `^` operator (commonly indicating exponentiation, but not a standard SMT operator), might not trigger the usual optimisations made during solving. On inspection of the implemented fix[13], it does seem the case that this simple example was an indicator of something missing in the decision procedure chosen by `Z3`.

This example is a good indicator of the difficulty of handling fuzzed bugs, and their interaction with the SUT. The test case seems simple and harmless enough, even under the consideration that this is a grey-zone. However, `Z3` is a mature project, and the fact that this issue never surfaced leads to the question of whether it is useful for the solver to consider such instances as bugs. The solution is simply beyond the capabilities of the solver, even if a human can easily recognise it. At the same time, because a human can so easily recognise it, the solver might be

[9] `https://github.com/Z3Prover/z3/issues/2238`, accessed 11th of October 2021
[10] `https://rise4fun.com/z3/`, accessed 11th of October 2021
[11] `https://github.com/Z3Prover/z3/issues/2604`, accessed 11th of October 2021
[12] `https://github.com/Z3Prover/z3/issues/4775`, accessed 7th of October 2021
[13] `https://github.com/Z3Prover/z3/commit/e2c1436cc8e17cbe93b18dc83d3af573dfea115e`, accessed 7th of October 2021

reasonably expected to provide an answer. But if no one is asking similar questions, then why should it?

Asan bug Bug 19[14] proved difficult to uncover. It was not discovered during the initial `MF++` process, but rather later on, during coverage-enhancing coverage experiments (Section 5.4.1). We observed a test that would crash on one machine, after a rather lengthy execution, and pass quickly on another machine. Further investigation uncovered the main difference was the compiler used for the test itself: one machine was using `g++`, while the other was using `clang++`. The problematic execution was only exposed with `g++`. Subsequently, due to such symptoms being potential indicators of undefined behaviour, we recompiled both the test and `Z3` with `clang++`, with asan enabled in both instances. This yielded a `heap-use-after-free` error. Upon communication with the maintainers of `Z3`, the root cause was identified as being a reference count mismatch due to implementation of `C++14` move semantics, and subsequently fixed. We managed to find another test case which exposed an identical issue, stemming from the same root cause, but in another part of the code[15].

`Yices2`

The two bugs we found in `Yices2` are at polar opposites in terms of complexity.

Bug 20, which we consider an API bug, showed an inconsistent result on a fairly small input. The function `yices_bvxor3` takes three bit-vector sort `term_t` inputs and returns a three-way xor. With a bit-vector size of 1, we found that creating three distinct `term_t` objects with values `0b1`, `0b1`, and `0b0` respectively, then passing them to `yices_bvxor3` in that order, we would see the result `0b1`. The expected result, as confirmed by applying both `yices_bvxor2` and the generic `yices_bvxor` functions is `0b0`. Upon report, we were advised this was a typo in the API, as `yices_bvxor3` would internally call the `or` function instead of `xor`. This bug has potentially been hiding away for a fair bit of time, presumably not being covered due to the potentially low use count of the specific 3-parameter xor implementation, when alternatives exist. When exercised, the bug would presumably be quite quick to manifest, even with very simple inputs. While in the end, we did manage to uncover a latent bug in the API, this particular testing instance shows the benefit of a more systematic, API-directed approach to generating tests with `MF++`.

Bug 21 on the other hand required a much more involved generation in order to expose an

[14] `https://github.com/Z3Prover/z3/issues/5435`, accessed 7th of October 2021
[15] `https://github.com/Z3Prover/z3/issues/5493`, accessed 1st of November 2021

Listing 4.4: Formula required to trigger bug 18 in `Yices2`

```
(push 1)
(assert (= x (ite (= x23 0) (* x59 x59) (mod (* x59 x59) x23))))
(assert (= z (ite (or (< (* (* x59 x59) (* x59 x59)) 0) (< (* (- 1) (* x x))
    0)) (* (- 1) x59) 6)))
(assert (= y (ite (= (* x x) 0) z (div z (* x x)))))
(assert (= (* (abs y) (ite (= (abs y) 0) 1 (div (abs y) (abs y)))) 0))
(check-sat)
(get-model)
(pop 1)
```

issue. Even after manual reduction, the formula itself looks quite hard to follow (Listing 4.4). It is a complex, non-linear formula, with two unbound variables `x23` and `x59` which `Yices2` computes as `unsat`. However, interestingly, if the `push` and `pop` commands are removed, the formula is found to be `sat`, and a model is provided (setting `x23 = -7` and `x59 = -2`). Of course, manually setting the variables to the given model shows indeed that the formula is expected to be satisfiable, and setting the whole formula within a push/pop context should not affect that satisfiability. The issue seemed to have been with computation of greatest common divisor, and did not require very much change to the codebase. Although it is particularly interesting why wrapping the formula within a push/pop context makes the bug be triggered, this is further evidence of how these solvers work internally, where small changes in how the formula is presented, which do not affect the formula itself, can affect the decision process.

`isl`

We have found many bugs in `isl`, primarily due to direct communication with the developers, including during the design process of the specifications. We shall discuss a selection of the more interesting issues we have found during `isl` testing, with the full details for each of the bugs being available at the corresponding report links.

The first discussion will focus on bugs 9, 10, and 11. All three bugs bugs were triggered by the same observable cause, namely an unexpected set equality check failure. All the three issues were flagged during the same experimental run, and we had assumed they might be due to the same root cause. This proved partly true, as all three bugs were failing due to improper *coalescing*, but seemingly, they were subsets of one another (meaning that potentially having reported bug 11 would have fixed both 9 and 10). However, in the order that they were reported, each report required further expansion of the prior fix. This slightly goes counter to our intuition, where we expected one fix to either fix all the issues, or other issues requiring a fix in an unrelated part of the SUT. It is hard to know when reporting bugs what the best

approach is, especially with a grey-box approach as was our case with `isl`, where we did not delve into the root causes of the failures we were seeing, but were sufficiently happy that the test itself seems to do something reasonable, but failing. Nevertheless, the end result is all three bugs managed to have been fixed, even if the whole process was less efficient.

Regarding the root causes of the bugs, they are to do with the *coalesce* [74] routine implemented in `isl`, which can be considered a sort of internal simplification operation. A high level explanation is that the routine looks at the shape of a set, and attempts to redefine it using fewer constraints. For example, if there is a set comprised of two adjacent subsets with an overlapping edge, coalescing might define the set as the union of the two subsets, which would be more efficient than having to include constraints for the common edge. There are numerous such heuristics implemented in the coalesce routine.

In regards to these particular issues, we managed to glean some additional information by directly asking a developer of `isl`. In regards to bug 9:

> Consider an integer set declared by the following two disjuncts (where x, y, z are parameters):
>
> $$0 \leq x, y, z \leq 100 \wedge 0 < z \leq 2 + 2x + 2y \tag{4.1}$$
> $$z = 0 \wedge x, y \leq 100 \wedge y \leq 9 + 11x \wedge x \leq 9 + 11y \tag{4.2}$$
>
> The constraint $z \leq 2 + 2x + 2y$ is valid for integer points in the second disjunct, but not for rational ones. Further, if we set $z = 1$, then the constraint becomes redundant with respect to $x, y \leq 0$. Since the constraint is not redundant for the first disjunct entirely, it means it is redundant (with respect to $x, y \leq 0$) on the hyperplane $z = 0$. Thus we assume the constraint is valid for integer points. (Tobias Grosser)

During the coalesce routine, we were advised that it is possible to include additional rational points as the two sets are coalesced together. However, it is essential that the number of integer points remains unchanged. Thus:

> (Bug 9) not only increased the rational points after coalescing but incorrectly included new integer points. While the first patch corrected our test case, it did so by making the overall routine more powerful, while relying on the assumption that

> redundant constraints have been marked correctly. (Bug 10) then exposed the fact that the polyhedron which did not exist before the coalescing routine was called, but those which have been created earlier in the iterative coalescing process, were not always scanned for newly redundant constraints.
>
> While the second patch addressed this instance of incorrectly updated state, (Bug 11) showed that the coalescing routine still relied on inconsistent state. The final solution implemented by the `isl` developers removed the earlier generalisation of the coalesce routine. (Tobias Grosser)

Overall, it seems there was some back and forth between initial existing issues present, test cases not fully exposing the underlying issues, and subsequent insufficient fixes from the developers.

In a separate instance, Bug 14 involves applying numerous MRs, especially DeMorgan's laws instead of direct union and intersection, in order to trigger a faulty `is_equal` check. We do note the need to issue `coalesce`, as well as `detect_equalities` calls at various points in the generation trace of the metamorphic variables. The function `detect_equalities` is another simplification procedure, which finds and removes implicit equalities. Inspecting the follow-up fix commit[16], it seems the main cause of this bug was again in the coalesce routine, due to "the disjunct with empty facet getting dropped entirely" (Sven Verdoolaege). The fix addresses this issue, adds further checks to ensure the output is as expected, and includes a regression in the test suite to check for this issue going forward.

`Omega`

Upon testing `Omega`, we noticed a high incidence rate of three issues, affecting roughly 70% of our tests. First, there were overall many segmentation faults, which we did not investigate further, but did make a note of, and reported the issue (Bug 1). The subsequent issues have to do with the API of `Omega`. While we could not find any prose documentation for the library, based on our experience with `isl`, and looking over the source code, we could infer what some of the functions would do. A function named `simplifyProblem` seems to simplify the internal representation of the expression being built. As such, we treated is as an identity function, being able to be called at any point, over any expression we construct. In our experiments, using this function would lead to an internal `Omega` assertion failure. This bug, noted as Bug

[16] `https://groups.google.com/g/isl-development/c/lpoPbJOms5c/m/DOpmlOOAAQAJ`, accessed 13th of October 2021

2, is not particularly critical, as we could not emit this particular function call (although we would be at its mercy if it were to be called internally at some point, and we are in the process of finding bugs anyway, which we would like fixed).

Bug 3 presents the most problems. As per metamorphic testing, we must be able to compare objects produced by the SUT. And the most simple way to do so is to check equality for sets we know are equivalent by construction (something we similarly do in `isl` via its provided `set::is_equal` function). There does not seem to be such a function readily available in `Omega`. Therefore, one avenue that seemed reasonable was to use the `MustBeSubset` function, as we would expect that two sets, which are the subset of one another, are equal. Unfortunately, we managed to synthesise tests where calling `MustBeSubset` on the same input function would lead to `false` being produced. Without knowing more about the implementation details, we do assume this should not be the case, and filed this as another issue.

We wanted to report these bugs, but the GitHub repository from which we obtained a copy of the library seemed abandoned, and the email link on the project's website[17] was also out of commission. One link that eventually was fruitful was the inclusion of `Omega` in the `CHiLL` [17] project. Upon reaching out to the current `CHiLL` maintainers, we were advised that we can file issue reports related to `Omega` to them, and were provided access to an internal GitLab system where to make such reports. Unfortunately, this system is not publicly available, but the related issues reported are present in Table 4.4. After a period of time, we were advised that support for `Omega` would be dropped, meaning that no fixes for these issues would be implemented. Due to the high incidence rate of the issues, we decided to not go ahead with further `Omega` testing. Nevertheless, it was a useful exercise to cross-define a specification for an existing domain (Presburger arithmetic) at an early stage of `MF++` development.

4.4.2 Coverage

During the evaluation of functional tests over the five valid SUTs (disregarding `Omega`, due to high rate of errors and lack of direct maintenance), we considered to further evaluate the utility of `MF++` by computing coverage achieved by our testing process versus the existing test suites of the SUTs in question. Thus, we performed a preliminary evaluation to compare coverage achieved by an execution of `MF++` for a given amount of time, versus the coverage for each of the tools in question.

Before that, we observe the approach of the five SUTs with respect to coverage. We note

[17] `http://www.cs.umd.edu/projects/omega/`, accessed 12th of October 2021

that CVC5 and Yices2 provide coverage reports alongside their nightly continuous integration builds. Furthermore, CVC5, Yices2, and Boolector include an option to compile with coverage-gathering features in their build systems. On the opposite side of the spectrum, isl and Z3 do not integrate any sort of coverage checks, and coverage features must be manually enabled in the build system. This can be done by modifying the appropriate environment variables (CFLAGS, CXXFLAGS, LDFLAGS) to include the --coverage flag (with a note that this must be done in the Makefile of isl at moment of writing).

Our experiment attempts to identify if MF++ can achieve any sort of additional coverage than the standard test suite of each SUT in turn. Thus, for each SUT, we perform the following sequence of operations:

1. Compile the SUT with coverage-gathering features enabled;
2. Execute the test suite and gather test suite coverage data;
3. Save the generated coverage data and wipe it from the build directory of the SUT;
4. Execute a MF++ run for a predetermined amount of time and gather achieved coverage;
5. Compute the delta between the two coverage logs thus obtained.

This experiment has been performed on a Docker 20.10.7 container, hosted on a Ubuntu 18.04.6 LTS host, with an Intel Core i7–6700 3.40 GHz CPU with 2×8GB DDR4 RAM. The compiler used was g++ 9.3.0. To gather coverage, we used gcovr 4.2, while to compute differential coverage, we used gfauto commit f796df52 from the GraphicsFuzz project[18] [27] The duration of the MF++ experiment was set at 20 hours for all SUTs, except for Boolector, where we used a time of 6 hours. The specifications of the SUTs were the latest version at the time of the experiment[19]. We used QF_BV logic for Boolector and Yices2, and QF_NIA for Z3 and CVC5.

The results of the experiment can be seen in Table 4.5. For each library, "Total Lines" indicates the number of lines in the library, as reported by gcovr, "Common" indicates the number of lines covered by both the test suite and MF++, "Suite" indicates number of lines covered exclusively by the SUT's respective test suite (and not the MF++ run), and "MF++" indicates lines covered during the MF++ run (and not by the test suite). Column "Commit" also indicates the specific shorthand commit used over which the experiment was performed. We

[18] https://github.com/google/graphicsfuzz/tree/master/gfauto, accessed 18th of October 2021
[19] https://github.com/0152la/SpecASTSpecs/commit/0880d1d3ad18f9ae76b56e88fd9649c323b16f88, accessed 18th of October 2021

		Coverage (Lines (%))			
SUT	Commit	Total lines	Common	Suite	MF++
isl	8073e847	75 574	8476 (11%)	42 399 (56%)	89 (0.12%)
Z3	f60ed2ce9	407 707	79 266 (19%)	102 993 (25%)	2161 (0.53%)
Yices2	3e61b88f	126 203	13 139 (10%)	65 069 (51%)	394 (0.31%)
Boolector	0783aa84	53 284	9848 (18%)	24 259 (45%)	81 (0.15%)
CVC5	861dba0ca	179 801	20 034 (11%)	115 332 (64%)	17 (0.01%)

Table 4.5: Differential coverage between SUT test suite and a MF++ run.

note that there was a slight mismatch, of about 1%, in total lines covered computed by gcovr between the two executions; this affects the number of common lines covered, but we believe the approximation is sufficiently accurate.

We observe that there is a small overlap in coverage achieved between the test suite and MF++ across the SUTs. The test suites of all SUTs have a moderate amount of achieved exclusive coverage, around the 50% mark, with Z3 notably lower, and CVC5 at the higher end (64% suite exclusive coverage, with a total of 75% total coverage achieved by the test suite itself). On the other hand, MF++ offers a rather modest relative exclusive coverage, under 1% across all SUTs. In line with the test suite results, we observe the largest additional relative coverage achieved over Z3, with just above half a percent, and the lowest over CVC5, with only 0.01%. This is in-line with how coverage is treated in the two projects—Z3 does not include it in its testing or building process, while CVC5 seems particularly interested in achieving higher coverage.

We note that the exclusive coverage achieved by MF++ over the provided test suites is minimal. Even the total coverage achieved by MF++, comprising of the sum of the values in the "Common" and "MF++" columns is rather on the lower side, hovering around ten to twenty percent. The main reason for this is that this experiment is an addendum to our main purpose of seeking bugs. The specifications we have prepared for these SUTs are not meant to maximize coverage, but rather to focus functional testing on specific parts of a SUT. We believe the test suites are meant to exercise wider sections of the SUTs, which explains their achieved coverage amount. Given these circumstances, we believe even this rather minimal coverage is valuable to investigate, considering the maturity and complexity of these libraries, and that this pilot experiment is proof that further work on this aspect of our approach is worthwhile. In combination with the automatic generation, and reduction of tests, a fully automatic infrastructure using our approach could be implemented to help polish test suites of software libraries, by indicating lines of code that could potentially be reached, but are not in the existing suites.

In addition, we also present the throughput of the above experiment in Table 4.6. For

Library	Throughput (tests/hour)	Timeshare (%)		
		Generation	Compilation	Execution
`isl`	256.46	67.49	30.84	1.66
`Z3`	161.96	36.62	8.15	55.19
`Yices2`	711.14	43.87	20.72	35.37
`Boolector`	605.64	36.54	20.77	42.66
`CVC5`	88.99	16.22	4.75	78.98

Table 4.6: Throughput of randomized testing using `MF++`

each SUT, we present the throughput, in terms of tests per hour, and how much of the time spent in total (namely, 6 hours for `Boolector`, and 20 hours for the other four SUTs) is split between generating test cases, compiling them, then executing them. We note that these numbers do not add up to 100%, likely due to system overhead. Overall, we observe a varied throughput, ranging from 89 tests per hour, up to 711. This is likely due to the theories used in the two edge cases: `Yices2` was applied over a bit-vector theory, while `CVC5` over an integer theory. Internal implementation differences might also affect total throughput. Generation does pose a rather substantial overhead, taking at most 67% of total experimental time in the case of `isl`. Potential further optimisations could be made to shorten this. Compilation and execution shares again are potentially heavily affected by the implementations themselves. It is particularly interesting how, for `isl`, the experimentation time is heavily affected by generation, and very little by execution. It does go in line with the expectation that the specifications we wrote are mainly finding bugs via the fuzzing aspect, rather than the metamorphic testing, which would explain the pattern we are seeing.

4.5 Related Work

There exist testing techniques specifically designed to work in the domain of SMT solvers. *Semantic Fusion* [78] is one such technique, where two equisatisfiable SMT formulas are *fused* together, to create a third formula of known satisfiability (due to the equisatisfiability of the inputs). The technique concatenates the two inputs, defines relations over variable pairs from distinct inputs, than randomly replaces instances of those variables using the aforementioned relations. These relations, named *fusion functions*, are similar to our SUT specification. They are defined by the authors of the technique based on the sort of the variables, while the choice of applicable fusion functions, as well as which variable instances to be replaced, is randomised.

Another technique bespoke for the domain of SMT solvers is *STORM* [51]. This is an

adaptation of *mutation fuzzing* [81, 16], a black-box fuzzing technique where some initial input data is mutated according to some rules. While the simplest mutation fuzzers flip bits in the input data, ones like STORM interpret this data, then mutate it via some domain knowledge. In this instance, STORM uses sub-formulas from the input, along with operations defined over the domain of SMT solvers, in order to mutate the input into a known-to-be-satisfiable test-case. Thus, an execution of such a test which yields `unsat` is an instance of a bug. The parallels with `MF++` are, again, the use of defined domain knowledge, in this case the process of mutating the input into a satisfiable output, and the operations available to apply over the identified sub-formulas.

Coverage is a metric used alongside testing in various works in the literature. For example, it is used to quantify how "good" generated tests are [13], or can be used as a feedback metric to adapt the generation of automated tests. Maximising line coverage is generally a good indicator of a reasonably solid test suite. However, a more advanced approach would be to identifying *path coverage*, which aims to find distinct execution paths in the program. There is some work attempting to maximise path coverage in SMT solvers [33], but this metric seems to be a fair way off maturity, as we are not aware of any practical tools computing path coverage in a given SUT, and it might be impractical due to the sheer scale of possible paths available. Nevertheless, for core, critical components, having different tests that exercise different execution paths might be a consideration.

4.6 Summary and Future Directions

In this chapter, we gave a brief presentation of all the libraries that have been tested with `MF++`, as part of our evaluation campaign. We delve into how to write effective specifications, and how we identified similarities to introduce abstract specifications. We present bugs that our implementation has found, and delve into deeper explanation of a number of particularly interesting bugs. These results present only a taste of the strength of `MF++`, due to time limitations, and the quality of the written specifications, which could be improved upon with better knowledge of the APIs.

Future Work The search space for SMT solvers is particularly large, and the work done here, while showing results, is just breaking the ice. Beyond the multiple other logics that would be available for us to use (each needing a separate, distinct specification), we could experiment with the large variety of properties and flags available for each solver. Of particular interest might

be the string and floating point theories, due to their rather sparse use, to our knowledge, in real-world scenarios. There are also a huge amount of parameters that can be set for each solver (`Z3`, for example, provides around 600 parameters, leading to numerous more combinations). The only blocker here is time, compounded with the need to understand the intricacies of both the new theories and parameters (as well as parameter combinations), and particularly the validity of written specification for new theories, or certain combinations of parameter values.

Beyond that, our specifications currently have one main type focus, either a bit-vector or integer sort type, supported by second-class MRs defined over boolean sort terms. There might be scope to add some more internal types (such as arrays, or functions), or any other elements that we might have missed as we were perusing the relevant documentation.

Due to the unified specifications, there could be scope to writing a DSL, which could be used to write domain-specific specifications, which could then be linked to as many back-end SUTs for that domain as desired. This could also be used to perform differential coverage—by using the same base DSL and same generation parameters, `MF++` would be able to generate a test that should be expected to have the same observable execution across SUTs (assuming there is no non-determinism in the particular domain tested; for example, a non-linear test could be solvable by one SMT solver, but computed as `unknown` by another). In addition, this process could be used in a performance testing manner, if one particular SUT shows vast performance improvement over another for the same conceptual input test.

The domains of the tested SUTs are rather similar, and can be considered numerical libraries. However, we expect our approach to work "out of the box" for any domain, minus potential infrastructure aspects in some libraries that we have not considered, and cannot be handled by our current infrastructure. However, that should mainly be a problem of engineering. Some of the considered domains to extend `MF++` testing to are graphics libraries (e.g., cairo [20]), filesystem libraries, and compression libraries.

[20] `https://www.cairographics.org/`, accessed 1st of November 2021

5 Automated Test Case Reduction for MF++

Reducing automatically generated test cases is an important final step in the testing process, ensuring that tests are high-quality, understandable, and clearly showcase the problematic input required to trigger a buggy behaviour. While manual reduction is a potential approach, where a human with some knowledge of the SUT can attempt to remove unneeded code, automatic test case reduction greatly increases the throughput of inspecting failing tests, and removes much of the routine simplifications that need to be done, while still maintaining the core of the test which exposes a potential bug. Another aspect is the property of fuzzers to find duplicate bugs (i.e., multiple tests which trigger failures due to the same root cause). Automatic reduction helps a human more quickly identify such instances of tests triggering the same buggy behaviour.

While generic automated test case methods exist, they do not (and of course, cannot) ensure that the properties of interest in our generated tests are not reduced. This primarily includes the metamorphic checks, which ensure that minimised test cases maintain an oracle. Further, implementing a bespoke approach to reducing automatically generated test cases can be much more efficient both from the point of view of time taken to reduce tests, as well as amount of source code being reduced. During the development of `MF++`, we have kept in mind the potential eventual need of reducing our tests, and ensured that generated tests have a systematic structure, to make them more amenable to a custom reduction approach.

We evaluate the utility of our reducer by performing a campaign of enhancing test suites of SUTs targeted during the functional testing process. By combining randomised metamorphic testing with automated test-case reduction, we can say this approach represents a novel method of coverage-guided test generation. Our initial coverage results (Section 4.4.2) shows that there is some potential for finding additional coverage via `MF++`. In addition to providing additional coverage, we note that, by construction, these tests come with an in-built oracle, further checking correctness and exercising the SUT in a meaningful way. An automatic reduction process will then be able to greatly reduce the manual effort required to put these tests into a human-

friendly format, with a low amount of further polishing making them of high enough quality to be included in test suites of mature libraries.

5.1 Motivation

When dealing with fuzz testing, or any sort of randomised testing, the core focus is maximising the efficacy of generated tests, in terms of which parts of the SUTs the tests exercise and how they do so. The presentation of the test does not matter at generation stage—the test will usually be consumed automatically, alongside thousands of other tests, and only once a test triggers some observable failure state is the test meant to be inspected further. As such, the readability of each individual test is an afterthought compared to the utility.

However, at some point, human intervention is required to inspect automatically generated tests, primarily to identify the cause within a test which triggers a failure state. We note that this failure might either be due to an actual bug in the SUT, or a malformed test case (due to an invalid MR, or a misuse of the SUT's API, for instance). In the case of `MF++` in particular and the domains chosen to test (namely that of SMT solvers and Presburger arithmetic), there is no immediate feedback when `MF++` generates a semantically invalid or not well defined input. We receive feedback from the SUT at the time when we explicitly query it, which will not indicate that we perform some operations over formulas of incompatible sorts, or which formulas within our API invocation map to those specific operands. Compounded by the fact that user-supplied information lies at the core of `MF++`, via provided specifications, such errors are quite likely. A reducer would thus hone in quickly on such instances where a specific MR is the sole cause of a failing test case.

Finally, the tests we generate are large by construction, as larger test cases are more likely to trigger bugs (discussed in Section 3.3 of the `Csmith` paper [80]). Generally, a bug will be triggered by a few select sequences of operations from within that large block of code. While removing the benign source code is mostly mechanic, it is quite laborious, and might even be tricky at times to follow data flow, due to the fact that the tests are not generated for human consumption. Thus, automatic reduction helps with removing the routine reduction required by humans, which should lower the overall time required to spend on a test to make it presentable.

5.2 Reducing MF++ tests

As mentioned, reduction was a feature that was desired to be implemented in the testing pipeline for MF++ tests. While it was not a core feature that directly shaped the design choices within MF++, certain decisions were take to ensure the design of MF++ was amenable to eventual reduction. We kept an abstract view of a generated test case in mind (by mainly having constructs allowing us to identify where expansions to the input template file have been inserted, as discussed in Section 3.4.3).

Thus, we defined four reduction opportunities which can be applied to our generated tests, by essentially identifying expanded code, and removing it entirely or in part. This approach improves the performance of the reduction process by ensuring that only reducible code is attempted to be removed by the reducer, and that if a reduction can be performed, it will leave the test case in a well-defined format. On the other hand, it does mean that the test might be able to be reduced further, but if we consider the user-provided template file as a ground truth, we believe that the template should suffice as potentially the smallest potential version that a given test case could be reduced to, which would be the case if there is an error within the template file itself. We expect that the delta between the template file and a reduced test case by MF++R to be sufficiently small to allow an expert in the SUT to identify where the fault lies.

5.2.1 Opportunites for Reducing MF++ Tests

We shall now discuss each of the four reduction opportunities implemented in order. We distinguish three reduction opportunities at the level of metamorphic variants—**Variant Elimination**, **Sequence Shortening**, and **Recursion Folding**—with one applicable primarily to the fuzzed input variables—**Fuzzing Reduction**. We further specify that the order in which we shall discuss these recursion opportunities is important, as we shall start with opportunities which have the most effect in terms of reduction, and will potentially subsume specific instances of subsequent opportunities (i.e., a variant elimination is expected to remove multiple sequence shortening and recursion folding opportunities, but not the other way around).

First, we provide an abstract overview of what a generated test case looks like, and where each of the reduction opportunities apply. This diagram is presented in Figure 5.1. From the template file provided by the user, we distinguish three main sections in the eventual generated test case: the recursive functions generated as part of metamorphic variant generation, fuzzed variables to be used as input for creating the metamorphic variants, and the metamorphic

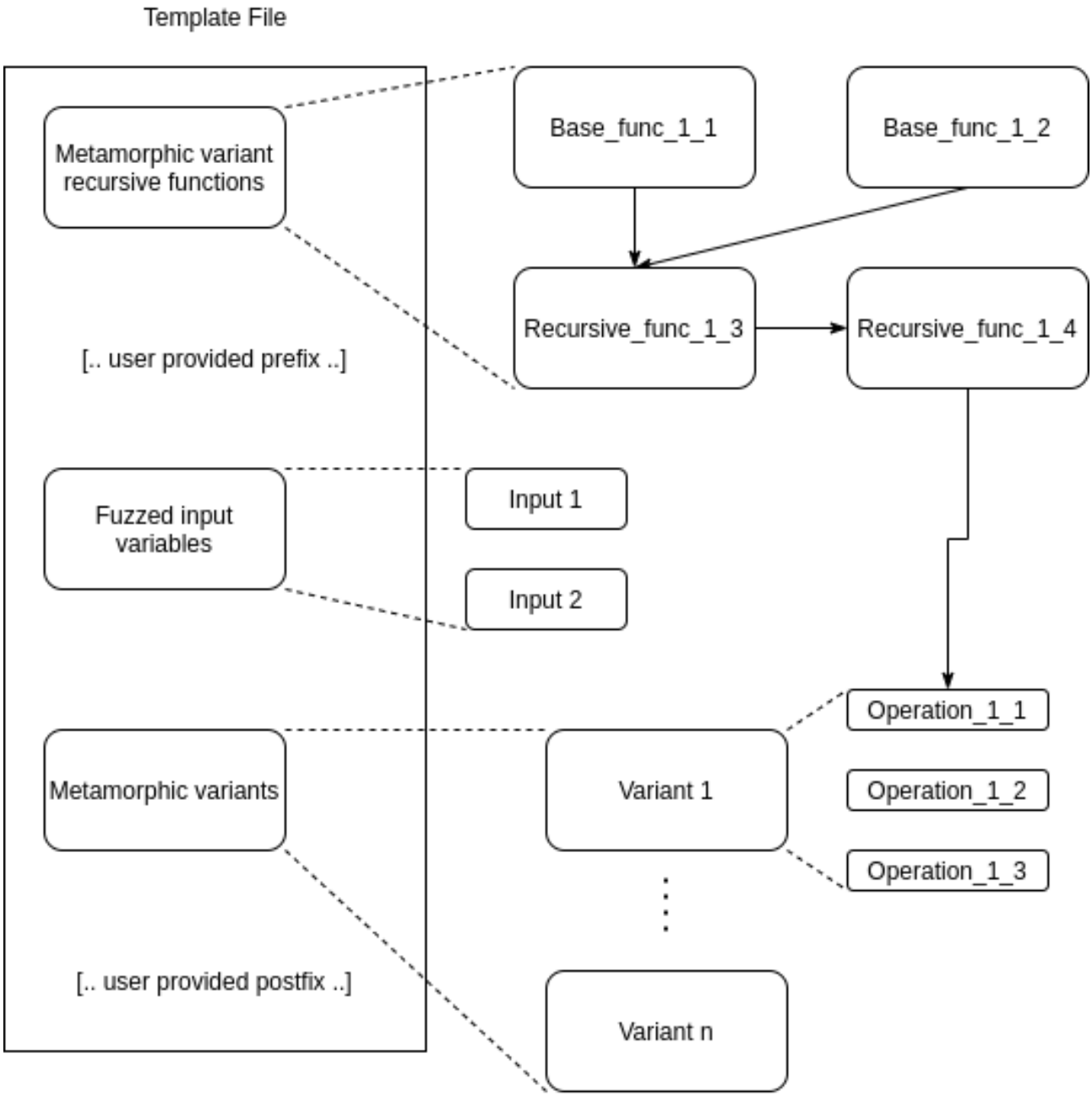


Figure 5.1: Abstract view of a MF++ generated test case

variants themselves. We can further identify the following disjoint (i.e., there is no dataflow between subsections within the same top-level section) subsections in our top-level splits:

- For the fuzzed input variables, we can identify sections of code which pertain to a specific variable (e.g., `Input 1`);
- For the metamorphic variants section, we can identify code pertaining to a particular variant (e.g., `Variant 1`); further, this code can be split into sections pertaining to one concretization of a high-level operation (e.g., `Operation_1_1`), as discussed in Section 3.2;
- Finally, the recursive functions sections contains sections pertaining to a concretization of one high-level operation of one specific variant; we can further identify an internal directed acyclic graph, which computes a result to be used by one such concretization; an example in our diagram is functions `Base_func_1_1` through `Recursive_func_1_4` computing a value to be used by `Operation_1_1`.

We make a small note regarding the "User provided" sections. These are part of the template file which might contain initialisation or cleanup code required to correctly call the underlying API. The reducer does not touch upon these sections at all. Furthermore, the code visible to the reducer is tied into syntax produced by the generator, as well as semantics built in the reducer regarding how to correctly interpret and process the code.

We shall now discuss each reduction opportunity in turn, referencing this abstract view of our generated test cases:

Variant Elimination This reduction pass refers to individual metamorphic variants, indicated as `Variant 1` through `Variant n` on our diagram. The tests we generate include multiple such metamorphic variants, and we check in a pairwise fashion whether the expected properties hold. Thus, it is highly likely that one of these pair-wise checks fails, and the two metamorphic variants which are checked against one another expose a bug alongside their generation trace. Thus, a reduction opportunity can involve removing an entire variant, with respective checks.

Sequence Shortening One individual variants consists of a sequence of concretized high-level operations. One reduction opportunity is removing high-level operations in the sequence across *all* metamorphic variants. In our diagram, this would mean removing represented `Operation_1_n` functions, and other corresponding operations across the other variants (not illustrated). This reduction is possible due to the fact that our high-level operations are designed to work on one main API type (see Section 3.4.4), meaning that we

can remove any operation within the sequence and still have a well-defined sequence of operations, with valid inputs. Further, the reduction needs to be performed across all metamorphic variants, in order to keep the values of the metamorphic variants in sync (although it might be possible to distinguish between identity operations and modifying operations to further polish the reduction process).

Recursion Folding When selecting a concretization for a high-level operation, called via the aforementioned `placeholder` function (Paragraph 3.3.3) within an MR, the generator can choose either between a base function, or a recursion function. We can reduce such a decision to a choice of a base function. Again referring to our example, this would entail reducing, for example, `Recursive_func_1_3` to a base function, which does not require inputs provided to it. We note how this means that `Base_func_1_1` and `Base_func_1_2` would then become dead code—this shall be discussed in Section 5.2.3.

Fuzzing Reduction Simplifying fuzzed code has been done before, primarily via delta reduction [63]. For our reducer, we shall consider other options, in order to maintain the validity of the fuzzed code sequences without needing to check this via compilation. One property of the fuzzed code sequences in our tests is that they are directed acyclic graphs. Thus, we can always remove later instructions within the fuzzed code block, which we know will produce objects which will not be used later on, as each fuzzed code block is disjoint. One other approach to reduction, which goes back to interfacing with user-provided data, is to have the user provide a *base* value for a given exposed type the fuzzer is aware of. The reducer can then attempt to change the values of objects of such types to this base value. For instance, for a type implementing an arithmetic set, the base value could be the empty set.

5.2.2 Reduction Toy Example

To better illustrate the reduction process, and available reduction opportunities, let's consider the example in Listing 5.1. The first step is attempting to perform variant elimination. We distinguish three metamorphic variables (line 25 to 35), named $v1$, $v2$, and $v3$. Variant elimination means attempting to remove all instructions writing to a metamorphic variant. Opportunities to apply variant elimination are variants $v2$ and $v3$; we ignore $v1$ as it is a metamorphic reference variable which we use to perform the metamorphic checks (lines 32 and 35). A successful reduction of $v2$ would eliminate lines 29 to 32. Crucially, we note that we eliminate the check

Listing 5.1: Example toy test potentially generated by MF++, to be reduced by MF++R

```
1  int add_comm_2_1(int e1, int e2) { return e2 + e1; }
2
3  int neg_by_sub_2_4(int e1) { return 0 - e1; }
4  int iden_by_double_neg_2_3(int e1) { return neg_by_sub_2_4(-e1); }
5  int mul_comm_2_2(int e1, int e2) { return iden_by_double_neg_2_3(e2) * e1; }
6
7  int add_by_sub_2_6(int e1, int e2) { return e1 - (-e2); }
8  int mul_by_add_2_5(int e1, int e2) {
9    int p = 0;
10   int sgn = e2 > 0 ? +1 : -1;
11   e2 = abs(e2);
12   while (e2 != 0) { p = add_by_sub(p, e1); e2 -= 1; }
13   return p * sgn;
14 }
15
16 void main() {
17   int f1 = 5;
18   int f2 = f1 * 2;
19   int f3 = f1 + f2;
20   int i1 = f3;
21
22   int f4 = 12;
23   int i2 = f4;
24
25   int v1 = i1 + i2;
26   v1 = v1 * i2;
27   v1 = v1 * i2;
28
29   int v2 = add_comm_2_1(i1, i2);
30   v2 = mul_comm_2_2(v2, i2);
31   v2 = mul_by_add_2_5(v2, i2);
32   assert(v1 == v2);
33
34   int v3 = [..];
35   assert(v1 == v3);
36 }
```

performed at line 32. This is safe to do, as we have another check at line 35. However, to ensure an oracle is preserved, we then do not attempt to eliminate $v3$. It might be possible to eliminate $v3$ and include a better oracle, but we leave this to further manual reduction.

The next reduction type to attempt is sequence shortening. In this example, we observe there is a sequence of three operations: `addition`, `multiplication`, and `multiplication`. This step attempts to remove one of these operations across variants. Suppose we want to attempt to reduce the first multiplication, the second operation in the sequence. Concretely, this means lines 26 and 30. We note that this is feasible, as the metamorphic variants remain in-scope for the third operation in the sequence. However, we do not attempt to remove the first operation in the sequence, as that has the variable declaration built-in.

Following is recursion folding. If we follow the chain of recursion for the second operation for

$v2$ (line 30), named `mul_comm_2_2`, we notice it expands further, to `iden_by_double_neg_2_3`, which calls `neg_by_sub_2_4`. This chain of recursion provides two opportunities for reduction. Attempting to reduce the `identity` implementation, we must choose MRs defined within the spec which do not have further recursion. We expect such a potential implementation of `identity` to take one parameter as input, and directly return it, without further calls. Thus, the call `iden_by_double_neg_2_3` can be replaced with just $e2$.

Finally, fuzzing reduction is the last reduction step in our process. We identify two input variables in the example program, $i1$ and $i2$. Let's consider $i1$, which is fuzzed after a sequence of 3 operations (lines 17 to 20). Suppose we provide a base reduction value for the `int` type of 0. What the reducer does is attempt to assign all the int objects in the fuzzing sequence to that base value. In our case, it would attempt to do so over variables $f1$ through $f3$.

5.2.3 Reduction and Dead Code

We now discuss the relation between generated code sections, and how they relate to the reduction process. As the reader might have observed from the discussion above, code in one subsection might have been generated to support code in another. For example, a concretization of a high-level operation (`Operation_1_1`) exists only to be called as part of the generation process of `Variant 1`. Subsequently, `Base_func_1_2` exists only because it is called by `Recursive_func_1_3`, which again is dependent on `Operation_1_1` existing. Therefore, performing a reduction somewhere along this chain of dependencies means there might be dead code left in the reduced test case. For instance, in the example in Listing 5.1, eliminating $v2$ means the functions used to generate $v2$ (lines 1 to 14) become unreachable (i.e., dead code). If variant elimination over $v2$ is successful, we would like to remove these lines, to ensure they are not identified as future reduction opportunities, and to eliminate as much unneeded code as possible.

We mentioned before that the order in which the reduction opportunities are presented is important based on their effect on the test case. What we mean is that a reduction opportunity with a higher effect can make reductions with a lesser affect non-applicable, and transform initially live code into dead code by the reductions they perform. The reason is that a high-effect reduction will affect code with dependencies targeted by a lower-effect reduction. By removing the original dependency, the code targeted by the lower-effect one has no more meaning. Further, by the directed nature of this dependency, it is impossible for a lesser-effect reduction to make a higher-effect reduction non-applicable (e.g., removing a recursive function call

generated to concretize a high-level operation does not affect whether the whole metamorphic variant can be removed or not). Thus, we can identify another property of our reduction process: we can perform the reduction in order from highest-effect to lowest-effect, and the reduction process should yield the minimal possible example producible. Furthermore, once a reduction opportunity has been found to be successful (i.e., the property we are aiming to reduce for is maintained after the reduction has been applied), we do not need to visit prior high-effect reduction opportunities that have been attempted; their applicability should not have changed.

5.3 MF++R

We have implemented our ideas in a separate tool, named `MF++R`, which takes as input a test produced by `MF++`, and a script to determine whether intermediate tests between reduction applications are valid or invalid with respect to the reduction process.

We decided early to have `MF++R` as a complete separate tool from `MF++`—while `MF++R` is expected to work on test cases produced by `MF++`, it does so by working at the level of source code, rather then the in-memory representation created during the generation process by `MF++`. There are two main advantages of this approach. First, it keeps the two codebases separate, with separate issues (i.e., a bug in one of the tools is clearly from that tool, and not from somewhere along the fairly complex generation into reduction process). Second, we can modify the input of `MF++R` on demand, without requiring to run `MF++` to produce the in-memory representation. This point did prove rather useful as we were using `MF++R`, and wanted to make manual changes to the input files, while keeping the expected format correct. Of course, the main disadvantage is that we do require to re-parse information from the source code, which might be readily available in-memory during the final stages of `MF++` generation. However, this point can be balanced out by the fact that gathering this information again in `MF++R` allows us to organise it in a manner more suited for reduction, than perhaps how `MF++` organises data internally.

The `MF++R` tool is implemented using `clang` libtooling, similar to `MF++`. A simplified view of its internal workflow can be seen in Figure 5.2. The input program is taken and we first perform a validity check over it, to ensure the program correctly contains something of interest to be reduced against. This is done via an **interestingness test**. Then, we parse the source code via **opportunitiesGatherer** to identify all reduction opportunities. This means matching concrete code to the representation seen in Figure 5.1, and organising it internally based on what reduction type we can apply over that code. Next, we iterate over these identified reduction

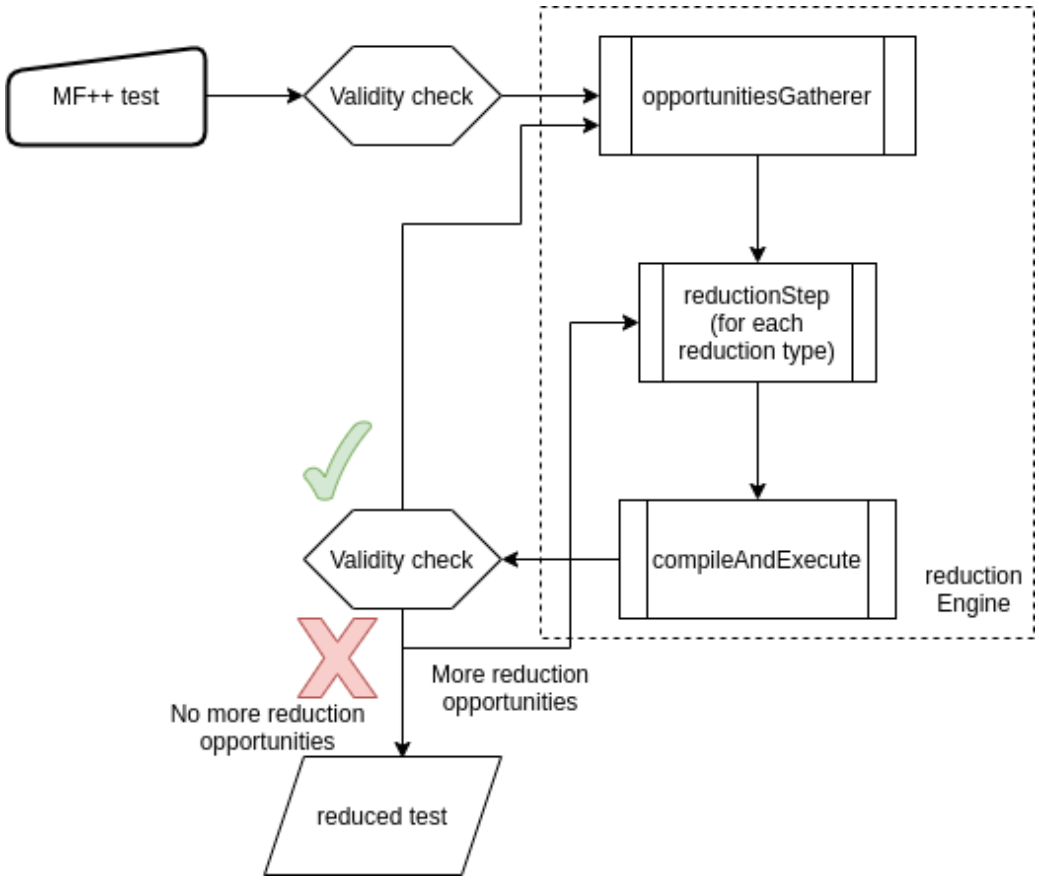


Figure 5.2: Internal workflow of MF++R

opportunities. One **reductionStep** comprises of one application of a reduction opportunity, which generally means removing or replacing the respective code sequence in some fashion. The reduced test case is then written to disk as temporary file, compiled, and the interestingness test is then used again to validate our reduced test. If the program is valid (i.e., the interestingness test finds that the property of interest, which might be an assertion failure for instance, holds), represented by the green tick mark, then we proceed back in the **reductionEngine** loop, starting with the **opportunitiesGatherer** again. If the reduced test is not interesting, then we proceed to attempt another **reductionStep**. Once all reduction opportunities have been exhausted, it means there is no more valid reduction to perform, at which point we emit the reduced test case to disk and complete the reduction process.

Interestingness test The purpose of this test is to decide whether a given test case is *valid*, with respect to some property of interest. This can range from various properties such as a particular `assert` being triggered by the test, the output of the test containing some particular substring of interest, a specific return code has been observed, or any other property that might be deemed of interest. In addition, to simplify the implementation of MF++R, the interestingness test is expected to also compile and execute the code, requiring primarily only the path to the

reduced test case to check.

In MF++R, the interestingness test is provided in Python format, and called from within MF++R. In addition to the path to the test case to validate, we also pass in a compilation script and a CMake configuration file to help with compilation. This ensures that all requirements are met for correctly executing against the desired SUT.

opportunitiesGatherer This step identifies the code over which our reductions can be performed, and logs all required information to actually perform each such reduction. In order to do this, there is some reference to information used to generate the given test cases. The following information is recorded:

- For Variant Elimination, we log instructions which write to variables with a name known to be generated by MF++ (by using the same specific formula used to generated these names in MF++);
- For Sequence Shortening, we can use the indices of the above information to identify the length of the sequence and which indices of the sequence to remove;
- For Recursion Folding, the process is slightly more involved. We first log all MRs declared in the corresponding SUT specification (which is available to clang due to including the header containing the declarations), distinguishing *base* MRs, namely those that do not have further high-level operation calls. For each instruction which writes to a variant, identified earlier, we traverse the corresponding function body, and identify function calls which are MRs. Any such function must have been generated due to a recursion point in the user-provided MR. We log each such function call, and the function within which it was found;
- For Fuzzing Reduction, we emit two dummy functions with empty body definitions. Both dummy functions take one string parameter, representing the name of the fuzzed variable, and is mostly used to distinguish pairs of dummy functions. These are used as delimiters of regions which fuzz objects. We log all instructions within the two regions, distinguished by the variable which is fuzzed within the region.

We note that we must perform this step after a successful reduction, as the internal pointers used to refer to clang objects are not valid across different file instances. As we must know where in the source code each specific component lies to perform the reduction (as we do so via clang's Rewriter), these pointers must be refreshed.

reductionStep Having gathered all the information above, we can now attempt to apply each reduction opportunity in turn. For each reduction opportunity, we modify the text of the test case appropriately, based on which type it is. We modify the code in memory using a `Rewriter` object, then emit it to disk as a temporary file, ready to be validated.

As reduction opportunities themselves refer to distinct parts of the code within the same reduction type, we can attempt to apply multiple reductions of the same type at once. Our algorithm takes all available reduction opportunities for a given type, attempts to apply half of them at once. If the reduction is not validated, then we attempt the other half. If that fails again, we attempt a quarter, and so forth, until we attempt a single opportunity at once. Once all opportunities of one type have been exhausted, we attempt opportunities of the next type of lesser effect, and the cycle repeats.

If a validation succeeds, we mark the reduction as successful, and begin the process again from the **opportunitiesGatherer**, using the newly reduced test case as input. If validation fails, and all reduction types have been exhausted, we note the reduction complete, and output the final, reduced test case.

5.4 Reduction with Respect to Coverage

With an automated test case reducer at our disposal, and having done a preliminary investigation of whether `MF++` produced tests could be used to obtain additional coverage (Section 4.4.2), we decided to investigate this approach further. The plan is to use `MF++` to obtain what additional coverage could be reached by our infrastructure (an exploratory phase), manually inject a fail state in the SUT to find instances of test cases which are able to reach the desired coverage (a coverage test gathering phase), perform reduction with respect to this injected test case, and obtain a small test case, exposing the new coverage, and including an oracle with a meaningful check. By manually injecting a failure state, we can reuse our bug-finding infrastructure, by pretending that a test case which triggers desired coverage is failing. While the injected fail state is guaranteed to be reachable (as it was originally randomly covered by an exploratory execution of `MF++`), there are two factors to consider whether the fail state can be reached again during the coverage test gathering phase: (a) the chance of randomly generating another test to reach the fail state is directly proportional to the amount of times the line of interest was covered during the exploratory phase, and (b) it might be the case that manually injecting multiple fail states might preclude certain fail states from being reachable, as execution would require executing code which now has an injected fail state. However, in our experience, we were

able to gather tests for all injected fail states, therefore we can assume coverage is reproducible when using the same version of `MF++`, and the SUT.

There are a few observations to be made about this process. First, the additional achieved coverage might be relatively small compared to the entire SUT codebase, which can easily span a few hundred lines. We believe additional coverage might be of interest, if even to provide an example of how it can be achieved, and to consider why the existing test suite does not provide that coverage. And finding additional coverage in large SUTs is not easy; not only is following the code hard, due to its size, identifying coverage *of interest* for a SUT does require some knowledge or intuition about the internal functionality of that SUT. The second point, introducing a failing state in the SUT, further requires knowledge of the implementation of the specific SUT. For instance, we found that `Z3` introduced its own set of assert-like functions, which would override the usual `assert` functions offered by `C`. In `Yices2` and `isl`, the `stdbool` header was not used by default, so we decided to use `assert(0)` as a failure point, while for `CVC5`, there is an `Assert` function. The potential explanation is that "assert" in these tools might be expected to mean the SMT assert, rather than the code-level check we expect. Finally, `Boolector` allows the usual `assert(false)` we prefer.

Finally, while the reduced test case is minimised, it generally can be manually reduced further. The primary impediment to better `MF++R` reduction is that it works at the structure of the test case, with some semantics derived from the provided spec. Further shrinking can be performed by using specific SUT knowledge (which, in terms of automatic reduction, would include some way of defining, interpreting, and applying SUT semantics during the reduction process). This might involve replacing the existing oracle with something more specific to the reduced test case, making the test case easier to read for a human. Of course, user-provided text in the template file, which is not visible to the reducer, can likely also be manually reduced. Thus, a manual reduction pass to further polish the automated test case is usually recommended.

5.4.1 Augmenting Library Test Suites

We wanted to put the above idea into practical application. We already had some preliminary results of what kind of coverage we could expect `MF++` to obtain for us across the five SUTs. We went ahead and contacted the maintainers of four of the five SUTs to discuss potentially including test cases exclusively for coverage in their existing test suites. We did not contact the maintainers of `Boolector`, as we have been made aware that it would shortly be succeeded by another tool, namely `Bitwuzla` [59]. We were advised that `CVC5` does use coverage as a metric

to guide the generation of their test suite (which might explain the particularly low additional coverage achieved by MF++), and such tests might have to integrate carefully with their crafted test suite. Compounded by the fact that the additional coverage was already low, we decided to focus on the remaining SUTs instead.

The maintainers of Yices2 and Z3 were particularly keen on receiving tests focusing on testing the API of their respective tools (which perfectly falls in line with the fact that MF++ uses the API of these SUTs to test them):

> We are absolutely interested in integrating your tests in Yices2. [..] API tests would be especially useful. [..] If you could focus on API and not convert [to] SMT2 that would be more interesting to us. (Bruno Duterte (Yices2 developer))

> Tests added to the examples/c++ directory [**Author note:** this is the folder where Z3 API tests are located] could be very useful and welcome. (Nikolaj Bjorner (Z3 developer))

The developer of isl was also interested in tests which focus on coverage, in a more general fashion. The first question was how these tests would be supported in each SUT. For Yices2, the developers gracefully implemented a special category of API tests, where we could add our own tests in, and would interface with their testing process. In the case of Z3, this endeavour sparked a discussion about the used continuous integration environment, as well as the lack of coverage information produced by Z3 itself (which would act as a good baseline to validate additional coverage). The end result was that we have helped implement a infrastructure to interface with C++ API tests[1], and helped add a continuous integration pass which gathers the coverage from the Z3 test suite[2]. isl proved to be a different type of challenge, and the infrastructure integration will be discussed alongside details of the coverage augmentation test integrated.

In the end, as a proof of concept, our work led to the addition of 21 new coverage tests: 10 tests for Z3, 10 tests for Yices2, and one test for isl (due to isl's primary interface being written in C and MF++ producing C++ tests, we translated one test as a proof of concept). We attempted to choose coverage in an interesting manner, focusing on internal routines in the solvers, and leaving things like user-facing API functions to be organically covered (e.g., we would try to avoid reducing with respect to coverage achieved in the definition of a multiplication function in the C++ API). However, we did not very deeply investigate whether the selected coverage is

[1] https://github.com/Z3Prover/z3test/pull/27, accessed 21st of October 2021
[2] https://github.com/Z3Prover/z3/pull/5451, accessed 21st of October 2021

of particular interest in the algorithm, as we do not have very deep knowledge of all algorithms used internally in these tools. For a surface level presentation of what we focused our coverage tests over:

- In `Z3`: non-linear arithmetic satisfiability, Poly DD package, emonics, non-linear arithmetic basic and order lemmas, and polynomial handling
- In `Yices2`: model evaluation, term and rewrite handling, `mcsat` (`Yices2`'s non-linear arithmetic solver) preprocessor, arithmetic operations using black-red trees, rational number handling, `nra` plugin, and `uf` plugin
- In `isl`: the coalesce routine.

We do not discuss more detail, as many of these features are specific to SMT solving, and their explanations are beyond the scope of this work. These descriptions were extracted from comments of file where coverage was achieved.

Integrating Coverage Tests in SUT Test Suites

We discuss in more depth the process of integrating our coverage tests in the test suites of the three chosen SUTs, as well as some particularities and observations during this process.

As mentioned above, for `Z3` and `Yices2`, we implemented the additional tests as a separate category of tests in their respective test suites. Then, the test suites were amended to include testing this additional category. Thus, adding a new test would involve simply moving the new test file in the corresponding directory (`C++` format for `Z3`, and `C` format for `Yices2`). With this setup, adding new tests in the test suites of these two SUTs involved performing GitHub pull requests.

During the process of adding coverage tests to `Z3`, a few events of note happened. First, when submitting our first coverage test, during the validation process, we noted distinct executions on two different systems. On one machine, the test would execute successfully within a second, while on another, it would fail with an internal `Z3` error within roughly 10 seconds. This behaviour was suspect, but initial debugging found no cause for the discrepancy. We submitted the test as a proof of concept, and the maintainers of `Z3` mentioned that a memory leak was exposed by our test[3]. Further investigation was able to correctly pinpoint the cause of the bug, which was a `heap-use-after-free` as reported by `asan`. Further, the discrepancy was due to the use of different compilers. When compiling with `g++`, the issue would not be manifested in

[3] `https://github.com/Z3Prover/z3/pull/5442#issuecomment-889503465`, accessed 24th of October 2021

the execution, but clang++ would expose it. We did not further investigate why this discrepancy happens internally, but this showed us that applying asan to our tests can be fruitful in finding deeper bugs. A subsequent coverage test was able to uncover a second bug, due to the same cause as the first, but coming from a different part of the code.

A second test of note was one that would compile and execute within a few seconds when Z3 was built in production mode, versus roughly 16 minutes with Z3 in debug mode. Production mode was used to enhance the reduction process (as hitting a failure point would instantly abort in production mode, versus waiting for user input in debug mode, before timing out), but coverage gathering is done in debug mode. We believe this might be an instance of a performance bug, due to the great difference between the two executions, with no change to the input test case. When mentioning the discrepancy to the Z3 maintainers, we were advised that this is normal, and we should just increase the timeout appropriately.

Further, as a consequence of implementing a coverage reporting system for Z3 (done in addition to adding our class of coverage tests to the Z3 test process), the developers of Z3 were able to identify parts of the code that had no entry points, and could be safely removed[4].

Adding coverage tests for isl proved a challenge, and we decided we would only submit one proof of concept test, then focus on the other SUTs. There were two main difficulties we faced attempting to integrate our test in the isl test suite. First, while we were using a C++ layer to isl, it was not an official one. For this particular proof of concept, we performed a manual translation from our C++ API to the native C API of isl, but that was mainly feasible due to the small size of the test required to trigger the coverage; we might not be so lucky with further tests. Second, the isl test suite is self contained and very carefully crafted. In our particular test, we were obtaining additional coverage due to the coalesce operation functioning over some input with derived rational dimensions. While our example was fairly small, the same coverage could be achieved by explicitly defining rational dimensions for the set, which we were unaware of due to unfamiliarity with internal isl functionality. In the end, the developer of isl integrated such a test, by expanding a list of input strings in the appropriate section of the test script. This followed a rather long back and forth, of us trying to understand and integrate our test. This whole process, while eventually successful, as the additional coverage was eventually integrated in the test suite, was fairly time consuming. If we had a better understanding of isl itself, or we could suggest additional coverage improvements to the developers, with a practical example to trigger it, that might be better for adding coverage to isl via MF++.

[4] https://github.com/Z3Prover/z3/pull/5483#issuecomment-899914061, accessed 24th of October 2021

	Reduction			Size (bytes)	
	Factor (%)	Time (s)	Attempts	Before	After
Min	10	13	8	2118	1899
Max	97	1110	789	358 044	10 438
Median	69	789	85	15 470	4521
Mean	61	257	151	39 783	4751

Table 5.1: Results of executing 18 reductions with `MF++R`

5.4.2 Evaluating `MF++R`

During the process of submitting these 21 coverage tests, we also practically evaluated `MF++R`.

As a general performance evaluation of `MF++R`, we look at the data in Table 5.1. This has been generated by applying `MF++R` over 18 of our 21 total tests (we were unable to replicate the exact infrastructure setup for `MF++R` and our SUTs for the remaining 3 tests). The reduction factor represents the percent by which the test has been reduced in terms of bytes (i.e., a factor of 10% means that the reduced test case comprises of 90% of the bytes of the input test case), while the reduction attempts is the total number of attempted reductions (both failed and successful). We observe a respectable reduction factor, around the 61% on average, with a median of 69%. This shows our reductions are effective at removing the bulk of the code. As we expect, the bug itself is concentrated in a few specific places, and the bulk of generated code is removable. The reduction time is acceptable, with an average of 257 seconds, but seemingly dominated by a few very quickly performed reductions. The median time of 789 seconds still keeps the reduction in a rather reasonable frame, but potentially more representative overall. However, considering the number of attempted reductions, and the fact that for each such reduction we must write the test to disk, compile, then execute, the reduction time is fairly reasonable — an average of under 2 seconds per reduction attempt.

Additionally, we also note that during our experience with applying `MF++R` over a number of different tests across our three chosen SUTs, we observed that certain reduction types would be more easily applicable to certain SUTs. For example, it seemed that fuzzing reduction was more successfully applied to `Z3` than to `isl`. This might be due, in part, to what the specifications for each SUT provide to the fuzzer. It also might be due that certain base default values chosen for specific types are better than others. However, we have not experimented with providing the fuzzer choices to reduce for each API type. This is mainly due to the current design performing an exhaustive reduction, and implementing choices would greatly increase the search space of the reduction.

Listing 5.2: Example of test case where the metamorphic check was reduced to a manually crafted check

```
1 term_t power_term = yices_power(yices_square(x), 2);
2 yices_assert_formula(ctx, yices_arith_leq_atom(power_term, x));
3 assert(yices_check_context(ctx, NULL) == STATUS_SAT);
```

5.5 Caveats

In this section, we shall highlight some specific observations made during the application of MF++R. These observations are more particularities that should be taken into consideration, rather than issues that affect the applicability of our method.

5.5.1 Coverage Arising from Fuzzing

By design, the tests generated by MF++ come equipped with their individual inbuilt oracle, namely the metamorphic checks applied across metamorphic variants, in order to ensure that the tests execute correctly. However, when reducing with respect to additionally achieved coverage, it might be the case that these checks are not necessary in order to attain this additional coverage. Indeed, as we make use of fuzzing across the generated test cases, it might be the case that we create an object with an interesting property, which triggers certain procedures in the system under test. If the desired additional coverage happens to be contained within such a procedure, the reducer can remove the built-in oracle, as it is not necessary.

While the main goal of our tests is to obtain additional coverage, we must also ensure that the tests actually perform some meaningful check, in order to expose a potential fault in the newly covered code. As such, we have augmented MF++R with an option to ensure that the oracles are not removed. We can easily identify which instructions comprise the oracles, as they are contained within the `checks` namespace of the `metalib` specification, and they are part of the concrete sequence of each non-basic variant. We ensure that we specifically exclude logging calls to functions within the `checks` namespace during the gathering process when the `--keep-checks` option is passed to MF++R.

However, even with this consideration, the main goal is to augment the test suite with examples that are easily accessible to the developers of the respective libraries. As such, while the test performs *some* meaningful action, this action is not necessary to contain the metamorphic oracle check, especially when the test might be simple enough to understand what it is doing at a glance. Consider Listing 5.2. This Yices2 example has been fully reduced (with additional manual reduction) until two APIs calls were deemed sufficient in order to achieved the targeted

coverage. In this particular situation, the usual check, requiring two metamorphic variables, would be beyond the scope of what is required to ensure the coverage is achieved. Therefore, we manually insert a hand-crafted check based on the contents of the test, to ensure the code exercised by the test remains correct. In this case, we check that the result of raising a free variable within the integer domain to the fourth power is greater than the value of the variable itself, which should hold.

5.5.2 Validity of Coverage

Beyond the question of whether the coverage we deem interesting to target is indeed important for someone with very deep knowledge of the internal algorithms and workings of our SUTs, there is also a question of how correct we compute coverage practically (especially when there is no baseline to compare against). This posed issues when attempting to generate tests for `Z3` in particular. Initially, `Z3` did not provide any coverage report in its nightly continuous integration builds. While computing what additional coverage is achieved by a test, we would update to the latest available version of the tool to validate that coverage achieved is maintained after reduction. At times, we would not find the coverage we would be looking for still being present. Further, as we would gather coverage via `MF++` multiple times, we would update the tool in-between these runs to ensure that we would find the most up-to-date coverage possible. We observed that coverage was very volatile in `Z3`. Combined with a high rate of daily updates to the project, this meant that we could not be certain coverage within our tests would be maintained even in the short term. In the end, we decided to fix a specific commit to compute coverage against, and stop "chasing master", due to this volatility. The decision was made based on the argument that if those tests were achieving this additional coverage at some point in time, they must be doing something interesting compared to the provided test suite, meaning they are useful as regression tests.

One other aspect to briefly mention is the difficulty of using existing tools to gather `C` and `C++` coverage (also briefly discussed in Section 4.4.2). While the tools we use are fairly mature, the methods by which they work are rather opaque, and hard to ascertain whether they function correctly, or in line with expectations. Of particular note is the rigid directory structure, with files expected to be in certain places relative with one another, that might contradict how certain SUTs implement their build systems.

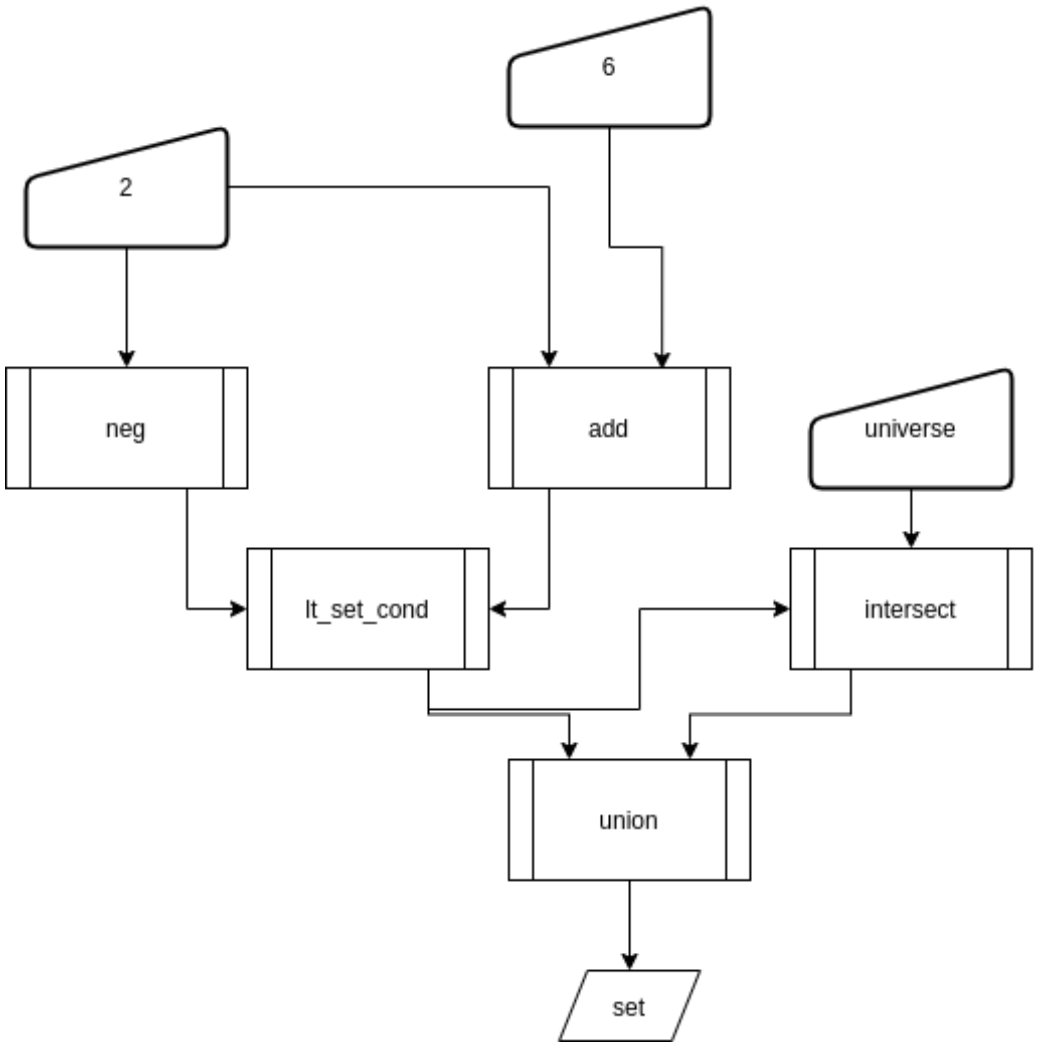


Figure 5.3: Example fuzzing process of a set in our example set library

5.5.3 Additional Potential Reduction Types

While our current four reduction passes are sufficient for removing large, unneeded parts of a given test case, there still are some obvious directions to improve the reduction process. These mainly stem from our experience of further manually reducing a test case, to make it more presentable and suited to be integrated in a real-world test suite.

The fuzzing reduction is in a fairly incipient state. Our experience shows that its utility greatly varies across tests and libraries. Furthermore, it seems to either be able to minimise large swathes of the code, or none at all, making it not that balanced. There is also a discussion to be had regarding performance, as fuzzing reductions are by far the most numerous reduction opportunities, but the wall-time duration of our reducer is low enough for this to not prove too much an issue at this stage. Nevertheless, it is worth keeping in mind, as the reduction process might grow in the future.

The most obvious direction to take the fuzzing reduction is to explicitly build the directed

acyclic graph (DAG) that we know each fuzzing sequence can be abstracted to, due to the way the fuzzing process itself works. This could be considered a completely distinct step than the current implementation of the fuzzing reduction, which only affects the data existing in the nodes, but not the structure of this DAG. Let's refer back to the example from Figure 3.2, reproduced in Figure 5.3 . We expect an explicit DAG would look similar to this figure, which is more or less a control-flow graph (CFG) of the specific sequence of code we are interested in, with some nodes existing outside the sequence, but only to be read from. One first improvement to be made is removing dead nodes. Suppose that our existing fuzzing reduction step would simplify the `add` node to be the base representation of a set, the empty set. Then, if the input node with value 6 is a node part of the code sequence, it can be safely removed. This does not happen automatically, as it is rather trivial, albeit time-consuming to do manually, but also straightforward once the analysis to produce the DAG has been implemented.

One other improvement is moving subtrees further up the graph. Consider the node `union` in our example. It takes two `set` variables as input, both produced by two subtrees. We can take any of these two subtrees and replace the `union` node with the respective base node of the subtree (in this case, either `lt_set_cond` or `intersect`). The idea behind this reduction is that **any** subtree within the tree can be used to replace any node further up, as long as the types are compatible, and make potentially large parts of the initial source code be unreachable, and thus candidates for deletion. Of course, this operation would be much more expensive than our current reductions, as for each node, we must identify all subnodes with a compatible type, and then attempt to replace them in turn. However, in absolute terms, the depth of a fuzzing sequence is generally not more than 10 nodes, meaning even a high complexity should not mean too high an increase in total wall-clock run-time.

Similar to the idea presented above, we can implement the same reduction opportunity at the level of MR recursion. If we consider a concretization of a high-level operation as the base of the tree, with edges linking to further concretizations requested in the body of the applied MR, then we can select a compatible concretization from within the respective subtree and move it up the tree, removing all intermediate recursive calls. We have noted instances where, using our **Recursion Folding** reduction, we would observe a single chain of recursive calls, with all the other high-level operation calls being reduced to base functions.

While this reduction is effective at potentially removing large parts of the recursive subtrees, it does affect the semantics of the program. A restriction that can be placed upon this approach is to ensure that the replaced base node and the chosen subtree node perform the same high-level operation.

5.5.4 Controlling Generation

Considering the reducer and the generator could be considered as one (it was an engineering choice to have two separate tools), a question arises on whether we could influence the generation process via feedback from the reducer. More specifically, suppose that we apply some statistical analysis over a given set of generated tests which identify bugs in the SUT. Could we use any of this information to guide the generation in a way to make it more likely to generate buggy code?

We believe that the answer is no; it is important during test generation to explore as much of the SUT space as possible. In this particular instance of employing feedback from already generated tests known to expose bugs, we could consider finding additional tests which expose the same issues useful — the root cause is unknown, and multiple ways of triggering the same bug might be valuable during the bug fixing process. However, overall we want to exercise as much of the SUT as possible, even during code paths that we know were correct previously, as it is unknown how they would react to different inputs. Furthermore, by using feedback from already found bugs, we would potentially hone our generator to just find more of the same bugs, and miss other bugs which might exist.

This is slightly related to the issue of generating small tests versus large tests. Our approach involves generating large tests, finding a bug, and reducing the test down to highlight the buggy code. One could question why wouldn't we start with a small test, and then keep on expanding that test into a large test, which could potentially preclude the need for automatic reduction. The answer is that such an approach has no well-defined termination: how do we know when we have expanded a test sufficiently to have triggered likely bugs? Or how do we know that a single additional expansion would not have been able to find a bug (supposing we expand tests up until a given limit)? In the case where we create a big test, we execute it and either a problem appears or not, and then the reduction process is applied as long as the bug persists and there are reductions to be performed — there is an explicit start and end. Using feedback, we similarly ask ourselves whether the bug we found were very rare, edge cases, or if there are bugs to be found around the "space" of these bugs. This question can only be answered by actually finding those bugs.

5.6 Related Work

One of the more common techniques to reduced fuzzed tests with is *delta debugging* [82, 63]. The method involves removing substrings of the input test program at the source code level, validating that the reduction maintains the well-definedness of the test case, and ensuring that a faulty behaviour still manifests. The removal is done with syntax of the underlying language in mind, such as attempting to remove full instructions, or lines within function bodies, and not fully chaotic removal that might leave function declarations incomplete. The main difficulty with delta debugging is ensuring the validity of the program is maintained. It might be the case that a reduction can introduce undefined behaviour in the program, which happens to manifest in the same way as the bug we are reducing against. This can be guarded against by using static analysers (e.g., Frama-C [42]), or semantic interpreters (e.g., KCC [30]). The approach of `MF++R` ensures that validity is maintained by removing code at the level of the `clang` AST, rather than substrings of the program. However, this is supported by being aware of the semantics of the input test program, as generated by `MF++`.

Our approach to reducing test cases with respect to a feature other than a specific bug is an instance of *cause reduction* [34]. Although the process by which we perform the reduction is by using failure points, similar to reducing against a fault in the program, the final result is that the reduced program, once the manually injected fault point is removed, achieve additional coverage, and the reduction process ensures that coverage is achieve along the reduction process. Other properties that might be of interest to reduce against is the amount of stress a test induces in the SUT, or perhaps some performance metrics.

Attempting to use reduction in order to improve coverage has been investigated before. Chapter 5 of a `GraphicsFuzz` experience report [27] discusses a similar approach to the way we generate tests for test suites of SUTs: differential coverage is computed between the target coverage suite (in the case of `GraphicsFuzz` and an execution of `GraphicsFuzz`, the `Vulkan` conformance test suite), failure points are manually injected in code that we wish to cover (via assertion failures), the tester is ran again, and failing test cases are those reaching desired coverage points, and marked for follow-up reduction. The main difference between the two approaches is that `MF++R` produces tests with oracles already included, as a consequence of them being generated by `MF++`. This provides users a starting point to either polish the existing oracle, or to be used as is, knowing that the test performs some meaningful test to ensure its correctness.

5.7 Summary and Future Directions

In this chapter, we discussed our custom approach to automatically reducing tests generated by MF++, which ensures intermediate reduction steps maintain the test case in a well-defined form. We discussed how we implemented this approach in a clang libtooling tool, MF++R. We discuss how our testing approach can be used to generate tests which achieve additional coverage compared to the test suites of our SUTs, then discuss how we used MF++R to add a total of 21 tests which achieve additional coverage to three SUTs. We pragmatically evaluate certain properties of MF++R during this coverage enhancing process, as well as devise some future improvements to the reduction process.

Future Work This whole process of generating tests, executing to find additional coverage, adding coverage failure points, finding test cases exposing additional coverage, then reducing could be scripted entirely. It would need some user-provided information, particularly the shape of the coverage failure points, and perhaps some changes to the current manual infrastructure we use, with gcovr and gfauto, but theoretically it should be a matter of just engineering. Fully automating this process would mean that we can leave the infrastructure run overnight and come back to a selection of potential coverage tests ready for manual evaluation.

As mentioned before (Section 5.5.3), there are additional reduction types that we have thought of while practically using MF++R, as well as shortcomings we observed in the currently implemented reduction types. This gives rise to two further directions: **(a)** attempt more reductions over other SUTs, and potentially observe further reduction types and more ways to fix current shortcomings, and **(b)** improve our current reduction process based on existing experience.

For computing coverage, we used the default line coverage metric provided by gcovr. However, other types of coverage criteria exist [84], including statement coverage (executing every statement in the program), branch coverage (executing all branches of all conditional statements of the program), or path coverage (executing every possible path along the program). Some are more feasibly to apply than others, such as path coverage being intractable for large software due to the number of possible paths. However, certain coverage metrics might be more suitable to certain SUTs than others, something which we could explore when performing reduction with respect to coverage.

6 Metamorphic Testing for Graphics Compilers

This chapter presents our experience of applying metamorphic testing to Open Graphics Language[1] (OpenGL) compilers. The work described here presents the foundation of what eventually became the **GraphicsFuzz** [26] tool. While the tool evolved beyond what is presented in this chapter, the core ideas and methods remain.

The work served as a precursor to the `MF++` project, and validated the use of randomised metamorphic testing as a potential technique to find bugs. The author of this thesis contributed original ideas, and was part of the initial prototype showcasing the application of randomised metamorphic testing for graphics compilers [28]. The ideas in this chapter provide a fundamental basis upon which the work in the rest of this thesis was built.

The idea of including identity transformations (Paragraph 6.3) was proposed and implemented by Alastair Donaldson, and is included as it was a main inspiration for subsequent for on `MF++`. A severe bug (Section 6.4.4) was found, investigated, and reported by Paul Thomson; the bug is included as an example of an important bug found via our metamorphic testing work.

6.1 Motivation

The main motivation behind undertaking this project was the expressed need by industry partners for methods of practically testing OpenGL drivers. This might be due to the advent of devices with embedded graphics processing units (GPUs), greatly increasing the hardware space where graphics and rendering are required to function correctly.

In addition to these requests, we had experience [46, 61] with testing Open Computing Language[2] (OpenCL), which is a similar API as OpenGL, but targets computing operations,

[1] `https://www.opengl.org/`, accessed 28th of October 2021
[2] `https://www.khronos.org/opencl/`, accessed 29th of October 2021

rather than graphics processes. While the two APIs are similar, coming from the same source, the test space is vastly different. Some major differences include some rather loose allowances the OpenGL spec allows, as well as inherent lack of accuracy due to the use of floating point numbers. Nevertheless, the domain of OpenGL comes with its own set of challenges, and we were unaware of a generally available testing tool for OpenGL available at the time. Extending our expertise with testing techniques to this new domain was a good personal motivator.

6.2 Testing Graphics Compilers

Setting out to test OpenGL, we must understand exactly what OpenGL is, and how it works practically. OpenGL itself is an API, accompanied by a standard and a compliance test suite. It is simply an idea, and the standard and API shape how an implementation of OpenGL should behave, while the test suite contains examples to check whether the finer details expected of an implementation hold. Therefore, there is no way of testing OpenGL itself.

However, a device, usually a GPU, that wishes to support OpenGL, must provide its own implementation of the OpenGL stack, according to the standard. Therefore, we define "testing OpenGL" as providing an infrastructure to test an implementation of OpenGL, while being aware of the OpenGL API which the implementation should provide. In more concrete terms, an industry vendor produces a piece of hardware which is expected to support OpenGL. In addition to the hardware, they must provide corresponding software to interface with the operating system (OS). Testing OpenGL for this particular piece of hardware involves testing the associated software stack as well. More practically, for a GPU, the corresponding graphics driver installed on the OS contains all the required software to execute OpenGL on the GPU. Particularly, each driver also contains a compiler for OpenGL, presumably tuned for the hardware available. Therefore, we distinguish an OpenGL *configuration* as a combination of **(OS, GPU device, GPU driver version)**. In the context of compiler testing, this provides a large search space of expected independent OpenGL compilers. We mention compilers here as we believe that to be the core component of executing OpenGL programs, and the one most susceptible to bugs.

While we mention there are OpenGL compilers, we did not discuss what exactly is being compiled. OpenGL renders an image by a series of successive operations over the data within the GPU. This is known as the *rendering pipeline*[3]. Each individual component of the pipeline takes the data from the previous action, performs its specialised transformation over it, and

[3] `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`, accessed 29th of October 2021

passes it forward. This specialised transformation is primarily defined by the user, as a piece of code called a *shader*. From the full OpenGL pipeline, we distinguish two processes:

- **Vertex Processing**, supported by a vertex shader, performs vertex-level transformations. At a high level, this means the geometry of the underlying object.
- **Fragment Processing**, supported by a fragment shader, defines what colour each pixel should be attributed. This basically defines the final appearance of the object, describing how it should be "painted".

There are other, more specialised processes, but these two are sufficient for basic rendering, and are what we focused on testing during this project. Primarily, it was the fragment shader that was the main focus. There could be potential to test other pipeline processes as appropriate, or enable/disable them at random.

With all this information, the basic ingredients required to perform graphics compiler testing are:

- a target **configuration**, comprising of a specific GPU device, OS, and GPU driver;
- a **host code** program, which communicates with the GPU, sending and retrieving required data;
- a **shader** for each pipeline process we would like tested; for this work, this would be vertex and fragment shader;
- a **test** to ensure that the image produced is correct.

The configuration is mainly dictated by available hardware, but there is work with ensuring that driver updates indicate that a new configuration is test. The initial implementation of the host code was done as a tool called **get-image**[4]. It made use of some third party libraries, (the OpenGL Extension Wrangler Library[5] (GLEW), and freeGLUT[6] (an open-source alternative to the OpenGL Utility Toolkit)) in order to abstract away the lower level details of using OpenGL. The purpose of `get-image` is to consume a vertex and a fragment shader, and display an image on the screen based on their execution on the GPU. We construct a simple vertex shader within the `get-image` script.

The more interesting aspects are the fragment shader and the test to ensure the image is correct. These are details which define the testing technique specifically employed by a testing

[4] `https://github.com/mc-imperial/get-image`, accessed 30th of October 2021
[5] `http://glew.sourceforge.net/`, accessed 30th of October 2021
[6] `http://freeglut.sourceforge.net/`, accessed 30th of October 2021

tool, and are similar to issues of generating inputs (i.e, fragment shader), and testing whether the output of the SUT (i.e., produced image) is within expectations.

We note that there are variations on OpenGL. The main variation can be considered the desktop version of OpenGL, which is also the most feature complete. An additional version is **WebGL**[7], which provides browser-level native support to execute a subset OpenGL code. This additional target requires its own specialised host code program, but beyond this additional engineering work, and ensuring that the used shaders are limited to the appropriate WebGL subset, the rest of the testing process remained similar. Finally, a third variant is **OpenGL ES**[8], which targets embedded and mobile systems, including gaming consoles, and mobile phones. While there are differences in features supported by the three variations, our core methodology is applicable across all of them, as long as we ensure that we generate tests within the boundaries of the capabilities of each target.

6.3 Metamorphic Testing with `GraphicsFuzz`

This section will focus on testing details specific to `GraphicsFuzz`.

We first discuss the issue of testing OpenGL output. We remind the reader of two properties of OpenGL: it makes heavy use of floating point numbers throughout the computation of what will eventually be the final rendered image, and the standard is loose at points, allowing GPU driver developers some leeway when providing their own implementations. This means that for a given rendering process, defined by a vertex and fragment shader, there is no ground truth image that can be computed. If the image is defined as a matrix of pixels with RGB values, then the acceptable output of a rendering process would comprise of a range of values per pixel, rather than a concrete number. This goes back to the oracle problem (Section 2.1)—we need a method of identifying whether the image produced by a concrete execution is correct.

This is a reason why metamorphic testing is a suitable testing technique for this domain. We cannot tell if an image produced by a set configuration is correct, but if there exists a reference image, and we generate metamorphic variants of the fragment shader used to generate the reference image such that the expected result is equivalent to the reference image, then we can use some comparison algorithm to check that the images are *similar enough*. In order to do this, we use the `OpenCV compareHist` function, with the `Chi-Square` method (suggested to us by Thibaud Lutellier[9]), and impose that the difference is under an empirically-derived

[7] `https://www.khronos.org/webgl/`, accessed 30th of October 2021
[8] `https://www.khronos.org/opengles/`, accessed 2nd of November 2021
[9] `https://ece.uwaterloo.ca/~tlutelli/`, accessed 2nd of November 2021

threshold for the specific tested configuration.

Having set the method of testing output images, the next step is finding interesting ways of generating them. The first question is whether a source of real-world fragment shader examples exists that we might use as reference for our eventual MRs. For this, we discovered, **GLSL sandbox**[10], which provides a rich offering of user-provided open-source fragment shaders. We lifted a number of 20 shaders, ensuring appropriate usage licenses were contained, which would serve as a repository for fragment shader sources.

Practically executing a selected fragment shader will then offer us a reference image to compare against. The final step is generating equivalent fragment shaders via metamorphic testing, to try and induce compiler bugs. There are two metamorphic transformations we implemented: *dead-code injection*, and *identity transformations*.

Dead-code injection The main transformation implemented in `GraphicsFuzz` is similar to the `EMI` [44] approach. However, instead of profiling code to find dead code (which is unlikely to happen in the fragment shaders we extracted anyway), we explicitly insert segments of code which we know will not be executed. This gives rise to two more challenges:

Ensuring injected dead-code is not optimised The compilers we are testing come with their own optimisation passes, similar to normal compilers executing CPU code. Explicitly injecting obvious dead code, such as an `if` statement with a `false` condition, or even something easily reducible to `false`, is likely to be outright removed from the executed code during optimisation. However, OpenGL allows the shader to read user-provided memory at runtime. This would ensure no compile-time optimisation can be performed in order to eliminate our injected dead code. The mechanism behind this is OpenGL **uniform** variables, which are read-only variables set at shader execution time. In our approach, they are set within the host code, after shader compilation.

Source of dead-code to inject Ideally, the injected dead-code should perform some sensible operation, to attempt and trigger miscompilations due to interesting control flow. Of course, fuzzing the injected code is reasonable, and might even be interesting if the fuzzed dead-code would involve undefined behaviour. This might stress the assumptions made by the compiler, due to the presence of potentially executable undefined behaviour. Nevertheless, we focused on injecting reasonable dead-code, and the source of this dead-code would be the other fragment shaders we lifted from GLSL sandbox, as they contain rea-

[10] `https://glslsandbox.com/`, accessed 30th of October 2021

sonable OpenGL code. We select a fragment of source code from the donor shader, ensure variable declarations are made appropriately (either by introducing new declarations from the donated code segment, or by renaming variables within donated code to make use of in-scope variables in the target shader), and obtain a fresh shader, which differs from the original one via a number of known-to-be unexecutable code sections. This injection should **not** affect the resulting image to an observable degree (again, the approximate nature of OpenGL might introduce small, machine level differences, due to things such as rounding).

Identity transformations Another method of producing equivalent shaders is via identity transformations. This involves replacing an expression with an *identity* of itself. Examples for integer expressions would include replacing an expression x with $x + 0$, $x * 1$, or $c?x : f$ (where c is a boolean expression evaluating to `true`, and f is an arbitrary, potentially fuzzed, integer expression). These simple transformations were sufficient in triggering some wrong image bugs [11] [12].

We note the similarities with `identity` high-level operations exemplified in Section 4.2.4. While `MF++` is able to use transformative MRs, we note that the domain of graphics compilers, with the precision issues encoded within, forces us to limit ourselves to identity transformations.

6.4 Evaluation

We have attempted to apply `GraphicsFuzz` to a total of 17 configurations, presented in Table 6.1. We distinguish three main types of configurations: testing OpenGL directly on a desktop machine, testing WebGL through a browser, done on a mix of desktops and mobile devices, and finally OpenGLES testing done on Android devices. We were able to identify at least one bug in *all* of these configurations, showing that OpenGL testing is in an incipient stage. These bugs also span a wide variety of vendors, which further emphasises that this is not a limited problem, but an issue with the ecosystem that should be addressed. OpenGL is becoming more prevalent nowadays with a plethora of consumer devices available. The sheer scale means that these bugs are expected to appear. However, the way they manifest might be particularly problematic.

In the following, we will showcase a number of severe bugs identified during `GraphicsFuzz`

[11] `https://github.com/mc-imperial/shader-compiler-bugs/issues/22`, accessed 2nd of November 2021
[12] `https://github.com/mc-imperial/shader-compiler-bugs/issues/25`, accessed 2nd of November 2021

ID	GPU	OS / browser / device	GPU Driver
		Desktop OpenGL (OS)	
1	AMD Radeon R9 Fury	Windows 10	Crimson 16.9.2
2	AMD Radeon HD7700	Ubuntu 16.04.1	AMDGPU Pro 16.40 348864
3	Intel HD Graphics 520	Windows 10	20.19.15.4501
4	Intel HD Graphics 520	Windows 10	21.20.16.4542
5	NVidia GeForce GTX770	Windows 10	373.06
6	NVidia GeForce Titan	Ubuntu 14.04.5	373.06
		WebGL (browser / OS)	
7	AMD Radeon R9 Fury	Chrome 55 / Windows 10	Crimson 17.9.2
8	AMD Radeon HD7700	Chrome 54 / Ubuntu 16.04.1	AMDGPU Pro 16.40 348864
9	Apple PowerVR GT7600	Safari 602.1 / iOS 10.1.1	n/a
10	ARM Mali-T628	ChromeOS 54.0.2840.101	n/a
11	Intel HD Graphics 520	Chrome 55 / Windows 10	20.19.15.4501
12	NVidia GeForce GTX770	Chrome 55 / Windows 10	373.06
13	NVidia GeForce Titan	Chrome 55 / Ubuntu 14.04.5	373.06
		Android OpenGLES (device)	
14	ARM Mali-T760 MP8	Samsung Galaxy S6 SM-G920F	OpenGLES 3.1
15	PowerVR Rogue G6430	Asus Nexus Player TV000I	OpenGLES 3.1
16	NVidia Tegra	Shield TV P2571	OpenGLES 3.2
17	Qualcomm Adreno 320	HTC One MR PN071110	OpenGLES 3.0

Table 6.1: Configurations tested with `GraphicsFuzz`

testing. We note that while these chosen examples paint a dire image of the OpenGL and WebGL ecosystem, these are the exception, and particular attention has been given to these examples due to their severity. Other types of issues identified are rendering the wrong image, or compiler crashes.

6.4.1 WebGL Blue-screen of Death

On configuration 1, we were able to induce a blue-screen of death by rendering a WebGL frame within a browser[13]. The blue-screen of death (BSoD) is a severe error in Windows operating systems, which indicates a non-recoverable OS-level error. The machine crashes, potentially corrupting user data, and even the hardware. Therefore, being able to induce a BSoD by rendering a webpage on a client machine, even if it requires a particular hardware configuration of the client machine, is a severe security issue.

We used a hand-written WebGL client, implemented in `JavaScript`, to render a specific fragment shader on a webpage. A client browser visiting the webpage with the corresponding

[13] `https://github.com/mc-imperial/shader-compiler-bugs/issues/28`, accessed 31st of October 2021

configuration might observe one of the following behaviours:

- The OpenGL process crashes, with a browser message "Rats! WebGL hit a snag!". The user is able to continue using their machine, even refresh the webpage, with the OpenGL process presumably restarting.
- The browser hangs, but the OS seems fairly stable. The browser can then be forcibly terminated via the Windows Task Manager.
- A BSoD is triggered, with the error `THREAD_STUCK_IN_DEVICE_DRIVER`, and the machine restarts.

The error message potentially points to a deadlock, or some concurrency issue. However, we did not investigate the bug further, primarily due to the fact that we had no direct access to the compiler contained within the device driver.

We were able to more consistently trigger the bug by using a client based on Almost Native Graphics Layer Engine[14] (ANGLE), which is an abstraction layer between WebGL and Direct3D, in order to improve OpenGL performance on Windows.

As the initial shader was fairly large, at 109KB, we attempted to reduce it to a minimal example that would still trigger the BSoD. The process of doing this is, of course, made difficult by the crash. At this point, we had an automated reducer implementation in a fairly initial state for `GraphicsFuzz` tests. We were able to automate reducing the shader using the following steps:

- on a separate machine, perform the reduction on the current version of the shader, and place it in a commonly accessible network location;
- fetch the reduced shader on the target machine, and execute it. After execution, write a file to the commonly accessible network location with some specific content indicating success. This indicates an unsuccessful reduction to the reducer, which can then proceed attempting other reductions, in a delta debugging [82] fashion.
- alternatively, if the machine crashes when executing the reduced shader, then it restarts. We set it up such that a script is executed on start-up. The script writes the commonly accessible file with content indicating a crash, prompting the reducer machine to update the current version of the shader to the newly reduced variant

[14] `https://chromium.googlesource.com/angle/angle/`, accessed 31st of October 2021

We essentially add an extra step between reduction phases, where the validity of the reduction is determined by the contents of the file which is commonly accessible by both machines. An update to this file acts as a signal by the target machine to the reducer machine. With this, we were able to fully automate the reduction process (although we monitored it, in case the machine would hang without a BSoD, or fail to execute the script proper on restart).

The issue was reported to AMD, and they confirmed it in their internal systems. However, they were unable to confirm the issue was present in the reduced version we provided them, only on the initial shader. This indicates that perhaps there were additional system properties at play which caused the reduced version to trigger the BSoD. Nevertheless, such a severe bug requires careful inspection. The issue was subsequently fixed in release 17.1.1 of the AMD driver.

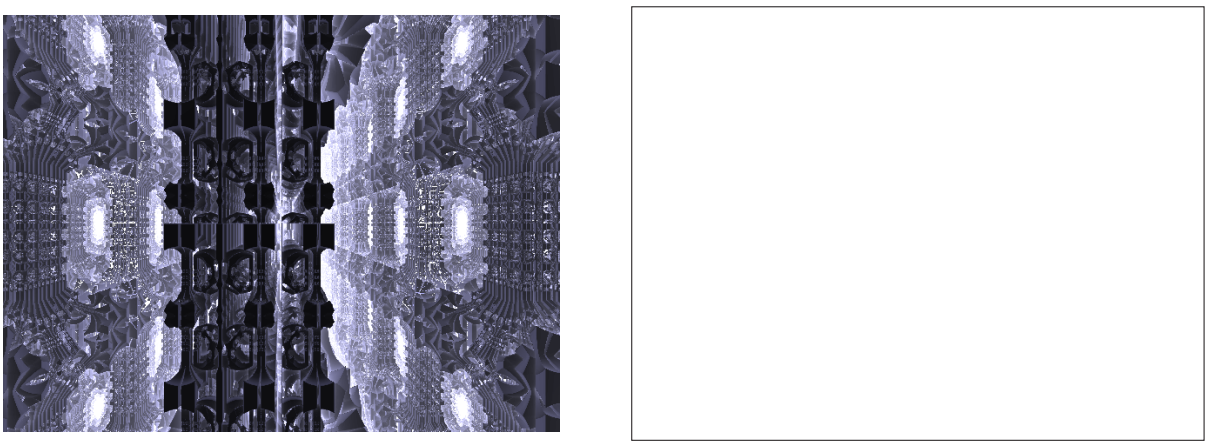
6.4.2 WebGL System Instability

On configuration 13, we were able to induce severe system instability by similarly rendering a shader on a webpage[15]. Using the same setup as above, rendering a shader inserted in a webpage accessible by any client, would show the following effects:

- a short, few second system freeze, with a message that the GPU driver cannot recover;
- a system freeze lasting between a few seconds, upwards to three minutes, after which the user is forcibly logged out; after logging in again, there does not seem to be any further system instability;
- the most common behaviour, a full system freeze, requiring a forced manual reboot.

Attempting to further inspect the issue, we observed that we were able to `ssh` into the affected machine. This would indicate that the freeze is at the display level, potentially the X server[16], which is the low-level interface between the display and Linux. Due to us being able to initiate a shell and perform commands on the shell, the system seems to be responsive, but the display completely frozen. Upon inspecting the workload, we noted that one of the 16 cores on the CPU was working at full capacity. Further, manually killing the browser process did not redress the problem. The underlying issue might be similar to the BSoD discussed above, but due to the difference in operating systems, the symptoms might be distinct.

[15] `https://github.com/mc-imperial/shader-compiler-bugs/issues/46`, accessed 31st of October 2021
[16] `https://developer.toradex.com/knowledge-base/x-server-linux`, accessed 31st of October 2021



(a) Reference image

(b) Variant image

Figure 6.1: Example of wrong image produced, after unreachable `discard` statement was introduced

We reported the issue to NVidia, and upon checking the issue at a later time, with device driver version 381.22, we observed a brief hang in the browser, without any further system instability. This indicates that the core issue was potentially addressed, and that the shader itself contains code that stresses the rendering process nonetheless. The issue was mentioned in a common vulnerabilities and exposures bulletin release[17], as CVE-2017-6259.

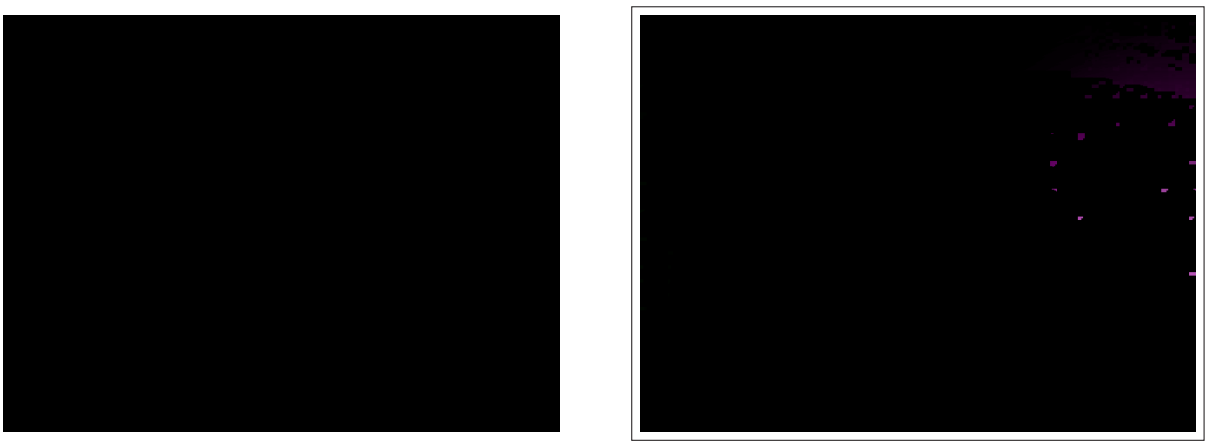
A similar issue[18] was identified using the open-source Mesa[19] OpenGLES implementation on an Intel processor with integrated graphics on the Google Chrome browser. The main differences are that the display freeze is not the most commonly observed behaviour, but rather equal parts display freeze, browser freeze, and apparent browser recovery after a period of freezing. However, attempting to render the shader multiple times can trigger the display freeze consistently.

6.4.3 Wrong Image Examples

We were able to trigger a variety of wrong render issues across the tested configurations, with rather minimal shader code changes.

On configuration 7, adding a `discard` statement (defined on page 120 in the OpenGL Shading Language specification [38]) after a `break` statement produces a blank image[20]. The original image can be seen in Figure 6.1a, while the variant image (Figure 6.1b) is a completely white image (borders have been added for clarity). Based on the language specification, we do not expect the `discard` instruction (which removes all data within the rendering buffers, thus

[17] `https://nvidia.custhelp.com/app/answers/detail/a_id/4525/`, accessed 2nd of November 2021
[18] `https://github.com/mc-imperial/shader-compiler-bugs/issues/67`, accessed 31st of October 2021
[19] `https://mesa3d.org/`, accessed 31st of October 2021
[20] `https://github.com/mc-imperial/shader-compiler-bugs/issues/29`, accessed 2nd of November 2021



(a) Reference image

(b) Variant image

Figure 6.2: Example of wrong image produced, after inserting a number of no-op statements

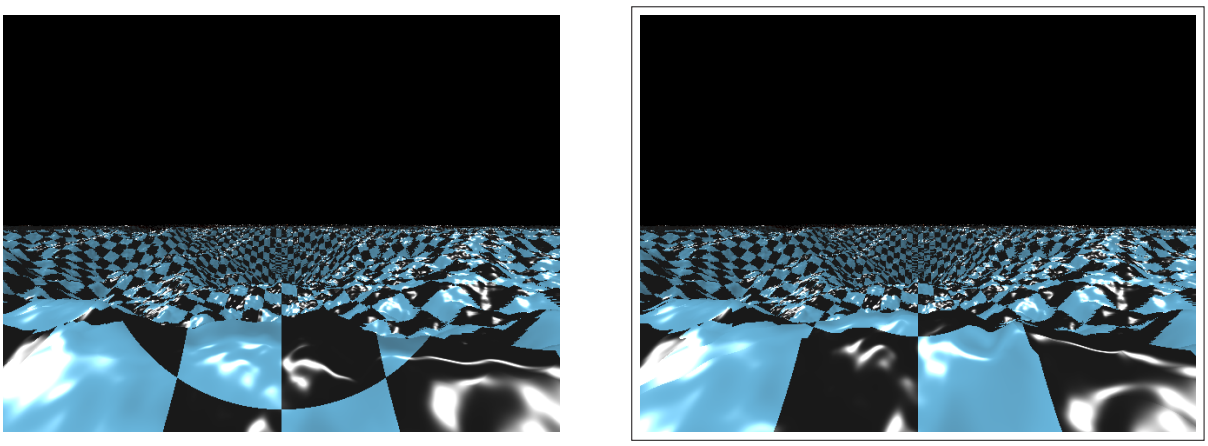
Listing 6.1: Code snippets introduced to trigger a wrong image render issue

```
1 for( int i = 0; i < 105; i ++)
2     for( int j = 0; j < 10; j ++) { }
3 [..]
4 if (injectionSwitch.x > injectionSwitch.y)
5     return;
6 [..]
7 if (injectionSwitch.x > injectionSwitch.y)
8     continue;
```

potentially explaining the blank image produced), to be reachable, as control flow should not go beyond a `break` instruction. However, due to miscompilation, it seems the `discard` operation has observable effect. This example indicates both the how easily some issues are triggered, as well as the difficulty in having correct compilers.

Another example, on configuration 1, produced unexpected changes in the variant image by inserting some instructions equivalent to no-op instructions[21]. There were a total of three code snippets introduced in the fragment shader code, which are shown in Listing 6.1. We note that `injectionSwitch` is a uniform (i.e., runtime) input 2-dimensional vector, and that the runtime value of the variable is set to $[0.0, 1.0]$. Therefore, the conditions in the two `if` statements are guaranteed to be false at runtime. Of course, the `for` loop similarly should have no effect on execution. The original image can be seen in Figure 6.2a, and the variant image is Figure 6.2b. There are some artifacts in the variant image which are not present in the original one. The reason might be that these shaders compute animations, while we capture a static snapshot of the render, and somehow the timing in the second image is affected by adding this additional code. Nevertheless, this is a clear render bug, as even without code optimisation, the runtime effect of the added code should be negligible.

[21] `https://github.com/mc-imperial/shader-compiler-bugs/issues/10`, accessed 2nd of November 2021



(a) Reference image

(b) Variant image

Figure 6.3: Example of false positive, believe to come from acceptable precision differences

Listing 6.2: Injected code leading to image differences within the boundaries of OpenGL standard acceptance

```
1 if(((injectionSwitch.x > injectionSwitch.y))) {
2         ray += vec3(1, 0, 0); }
```

The final example we present is an actual false positive, observed on configuration 6[22]. While the reference (Figure 6.3a) and variant (Figure 6.3b) are visually distinct, it is believed to be a consequence of the shader making use of trigonometric functions. Particularly, the standard says:

> Built-in functions defined in the specification with an equation built from the above operations inherit the above errors. These include, for example, the geometric functions, the common functions, and many of the matrix functions. Built-in functions not listed above and not defined as equations of the above have undefined precision. These include, for example, the trigonometric functions and determinant. (Page 94, OpenGL Shading Language version 4.40 [38])

While the injected code was minimal (Listing 6.2), and showed no indication of triggering expected precision changes, the fact that the original shader included trigonometric functions leads to undefined precision, and therefore the difference seen in our examples being within the parameters set by the specification. This example illustrates how fragile OpenGL testing is, and how much care must be taken to ensure that we very closely follow the standard to prevent precision differences. While this is comparable to avoiding undefined behaviour in `C` testing, the difficulty is unfamiliarity with the domain (i.e., background knowledge gives an intuition of

[22] `https://github.com/mc-imperial/shader-compiler-bugs/issues/51`, accessed 2nd of November 2021

Figure 6.4: Data leaking from a browser tab accessing a banking webpage

why certain `C` code is undefined behaviour, such as accessing an array out of bounds, but not why certain OpenGL function calls can cause imprecision, such as in this example).

6.4.4 Private Data Leak

The final example presented is an instance of an information leak. While the author of this thesis was not involved in identifying the bug, this issue is mentioned due to being found with the `GraphicsFuzz` infrastructure, and due to the severity of the issue. I thank Paul Thomson for his work on producing a proof-of-concept exposing this issue.

The issue[23] shows memory data leaking from a tab to another, potentially malicious, tab rendering a WebGL image. The issue was discovered on configuration 14. Shown in Figure 6.4, we see the result of attempting to render a WebGL shader in a tab, while a banking webpage is opened in another. Some information from the banking page can clearly be discerned in the rendered image, and a malicious actor could feasibly capture snapshots of this leaked data, and save them somewhere under their control. This shader could be included in a malicious webpage with a very small, potentially pixel-size viewport, such that the user is unaware of any malicious acts happening. The bug was eventually fixed via a firmware update.

[23] `https://github.com/mc-imperial/shader-compiler-bugs/issues/80`, accessed 31st of October 2021

6.5 Related work

To our knowledge, there is no similar testing technique applied to the domain of OpenGL compilers. The main idea of injecting dead-code in fragment shaders in this work came from the `EMI` [44] technique. The main distinction is that while `EMI` profiles code to identify dead-code over some set input, and then performs modifications in that dead-code to generate equivalent variants, `GraphicsFuzz` explicitly injects dead-code sequences in a compiler-obfuscated fashion, to ensure that these injected sequences are not optimised away.

There are some practical tools to help with OpenGL development. Khronos, the maintainers of OpenGL, provide a reference compiler[24], which can potentially be used as a rudimentary differential testing candidate, particularly for validating workflows and obviously incorrect shader outputs. There also seems to be a third-party GLSL (the language used to write shader code) validator[25], but we have not assessed its efficacy. Nevertheless, such a validator would only focus on syntax errors, rather than the functional errors `GraphicsFuzz` is able to detect.

6.6 Summary and Future Directions

In this chapter, we discuss the precursor to the `MF++` project, namely `GraphicsFuzz`. We discuss general challenges to testing OpenGL, and the suitability of metamorphic testing in overcoming those challenged. We focus on the initial ideas paving the core testing methodology of `GraphicsFuzz`, particularly the idea of injecting dead-code from donor shaders, to ensure the metamorphic variant comprises of interesting control flow. We provide examples of a number of interesting bugs, with potential security vulnerabilities, exposing the power of misusing OpenGL, and the importance of functional testing for security.

Future Work Additional work has been done in the `GraphicsFuzz` project beyond the involvement of the author of the thesis. This includes adding more identity transformations (such as transforming a float f into $(f + 0.0)$ or $(f * 1.0)$), employing automatic test case reduction methods for generated tests, and expanding the targets of testing beyond OpenGL and WebGL.

There is room to expand the scope of OpenGL testing. Currently, `GraphicsFuzz` focuses on fragment shader bugs, but we are unaware if there are special routines implemented in compilers for other pipeline processes. Expanding the shaders exercised by `GraphicsFuzz` is a potential feature improvement, but would come with its own challenges of finding appropriate sources of

[24] `https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/`, accessed 30th of October 2021
[25] `https://github.com/felixpalmer/glsl-validator`, accessed 30th of October 2021

respective shaders.

The end-goal for this project would be full integration within the conformance test suite of OpenGL. That would perhaps involve an infrastructure where there is a more consistent generation process, to ensure a fair evaluation of all provided drivers.

7 Conclusion

In this work, we present a new approach to using metamorphic testing, which we name *metamorphic testing with high-level operations*. This approach implements some of the finer techniques of applying metamorphic testing, while interfacing with the user to provide some minimal examples of redundant operations. Lowering the required ingredients to applying metamorphic testing and framing then in a practical context will hopefully make the technique more attractive to library developers.

7.1 Contributions

This work makes the following contributions

Chapter 3 provides an explanation of applying metamorphic testing with high-level operations, presenting all the metamorphic testing features included (such as composition of metamorphic relations, and MR-level recursion). We describe what ingredients are required to be provided by the user to interface with a new library, potentially leveraging knowledge from adjacent domains. We describe a tool, `MF++`, providing an implementation of our proposed method targeting `C++` software libraries, which additionally incorporates elements of fuzzing, in order to automatically synthesise expressive test cases with in-built oracles for finding bugs in SUTs.

Chapter 4 presents practically applying `MF++` to 7 libraries in the domains of SMT and Presburger arithmetic. We provide an overview of the challenges and design choices made as testing with `MF++` of these libraries proceeded, while also exposing the needed knowledge to make effective use of `MF++` for desired SUTs by implementing strong specifications. We provide details about 21 found bugs, all of which were reported, confirmed, and fixed by the respective maintainers.

Chapter 5 discusses the implementation of a custom-made automatic reduction approach for `MF++` tests. The benefit of the approach is that it ensures tests maintain well-definedness

throughout the reduction process, while maintaining features of interest, such as the oracle contained in the generated test case. Ensuring that the oracle is preserved means that reduced tests perform a meaningful check, and could presumably be readily integrated in a test suite. The additional benefits of implementing automatic test case reduction, aside from minimising buggy tests, are: it allows users to quickly identify errors with their written specifications, and it can be used to reduce with respect to other properties of interest. Such a property is code line coverage. As an exercise in assessing the applicability of this approach, we reached out to developers of three SUTs, and ended up integrating 21 tests in their respective test suites to improve coverage

Chapter 6 details the an initial approach to applying metamorphic testing in the domain of graphics compilers, attempting to overcome the numerous challenges the field poses to testing techniques. We devised a new strategy of injecting known-to-be-dead code, which is not susceptible to compiler optimisations. Having a rich source of code examples allowed us to inject real-world code with meaningful control flow. Metamorphic testing was employed to derive a variety of mutants, which can be compared against the ground truth produced by the original code. With our technique implemented in the `GraphicsFuzz` tool, we were able to identify numerous bugs across 17 configurations, spanning all major GPU vendors. These bugs include critical vulnerabilities, such as inducing machine crashes, or personal data leaks.

In regards to the posed research questions from Section 1.1, the results can be are as follows:

1. **Is metamorphic testing with high-level operations effective at finding bugs?**

   We were able to identify 21 bugs across 4 libraries. While we were familiar with the libraries chosen to be tested, we would not consider ourselves experts, and thus believe the approach can prove even stronger in the right hands.

2. **Can metamorphic testing with high-level operations be used to improve existing coverage metrics?**

   While coverage attained in addition to the test suites of the tested libraries was minimal in our evaluation (Section 4.4.2), it is the case that we were able to find some additional coverage. We note that due to the nature of how user-provided specifications affect the testing process, testing could be guided to focus particularly on finding additional coverage (something we have not consciously attempted during evaluation).

3. **Could metamorphic testing with high-level operations be applied to generic domains?**

   The domains evaluated in this work are SMT libraries, Presburger arithmetic libraries, and graphics compilers. While all these domains are numerical in nature, the requirements to applying metamorphic testing with high-level operations are not limited to numerical domains (Section 3.2). To further strengthen this argument, we have worked on an initial concept of applying this technique to the domain of graphics libraries, and all challenges faced during this phase were in terms of engineering.

7.2 Limitations

There are a number of limitations and insufficiencies that we highlighted throughout this thesis. Here, we emphasise the more important aspects that might require further work or investigation.

Single-type Focus during Generation In order to simplify both the engineering required and the design of `MF++`, we limit the type of MRs for high-level operations to be identical. This ensures that intermediate values across the sequence of high-level operations are compatible, and the sequence can push data along the generation. We do include second-order high-level operations, which can function over other API types, but they are constrained to be generated within the recursion tree of a first-order high-level operation expansion.

This limitation conceivably reduces the search space of generated test cases, but it does ensure data continuity in the metamorphic variant generation sequence. More investigation needs to be done to evaluate how much additional bug-triggering potential this would allow, in relation to difficulty of implementation, and coherent specification design (in terms of ensuring that the user is prevented from providing a specification which might be ineffective due to how `MF++` works internally),

Diversity of SUT Domains The SUTs chosen to apply `MF++` over cover two rather similar domains: SMT and Presburger arithmetic. They are both numerical domains, and challenges in one can be applicable to the other. Furthermore, the numerical nature of the domains potentially makes finding MRs easier than other domains. However, the method of applying metamorphic testing with high-level operations is a general concept, and can, in principle, be applied to any domain and in any programming language. The best evidence for generality should be practically applying `MF++` over more domains. We have considered some other do-

mains, such as graphics libraries, filesystem handling, or compression libraries, but have not acted on familiarising ourselves with these domains. There is an initial implementation of the `cairo` [1] graphics library.

7.3 Future Work

Potentially the most important next step for this project is to "let it loose in the wild": we could reach out to various library developers to present our technique and its requirements, and gather feedback about its ease of use, observed effectiveness, and a number of other metrics. The tool is in a mature enough stage, but development has been made with domain experts in mind, which is something we have not evaluated throughout the development of this project.

In addition, there are a number of unexplored ideas that have arisen during the development of this project that we have not acted upon due to time constraints.

First, we could apply a sort of *differential testing* approach, considering we test multiple SUTs in the same domain. This is further made possible by abstract domain-level specifications that we have devised. The core idea is that we have specifications for domains implementing common expected features that SUTs within those domains should have. With this abstract view, we could generate a domain-level test, which could then be linked to the an API of a SUT of interest. Having multiple such interpretations allows us to essentially execute the same program in different SUTs and compare results. We would expect a discrepancy to be caused by a bug in one of the SUTs. This approach is slightly orthogonal to metamorphic testing, but is applicable due to how we store SUT specifications for `MF++`, and could be seen as an extension of our approach, similar to our coverage augmenting work with the reducer.

Another tangential direction is *performance evaluation*. The MRs that `MF++` uses essentially present alternative implementations of the same operation. While there might be some internal optimisations, which would lead to similar, or even identical actual low-level operations, performance discrepancies are likely. Supposing a profiler exists to evaluate the performance metric of interest, we could use the MRs in `MF++` specifications in two ways. First, for a given high-level operation, we can evaluate which MR (or implementation) is best suited for the metric of interest, and potentially investigate at a low level why the discrepancy exists, to be included in other alternative implementations. Otherwise, we might do this across SUTs within the same domains. Two similar implementations in different SUTs might express highly different performance metrics, which might be due to some internal pathology in one of the SUTs.

[1] `https://www.cairographics.org/`, accessed 28th of October 2021

Bibliography

[1] Fuzzing#types. URL: https://en.wikipedia.org/wiki/Fuzzing#Types.

[2] libfuzzer – a library for coverage-guided fuzz testing., 2022. URL: https://www.llvm.org/docs/LibFuzzer.html.

[3] David R. MacIver Alastair F. Donaldson. An overview of test case reduction, 2021. URL: https://blog.sigplan.org/2021/03/30/an-overview-of-test-case-reduction/.

[4] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 1–11, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/73560.73561.

[5] A. Arcuri and G. Fraser. Evolutionary generation of whole test suites. In *Quality Software, International Conference on*, pages 31–40, Los Alamitos, CA, USA, jul 2011. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/QSIC.2011.19, doi:10.1109/QSIC.2011.19.

[6] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi:10.1109/TSE.2014.2372785.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[8] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_11.

[9] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakr-

ishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.

[10] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021.

[11] Raymond Boute. The euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14:127–144, 04 1992. doi:10.1145/128861.128862.

[12] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009. doi:10.1007/978-3-642-00768-2_16.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[14] Yuxiang Cao, Zhi Quan Zhou, and Tsong Chen. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. pages 153–162, 07 2013. doi:10.1109/QSIC.2013.43.

[15] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. Cross-checking oracles from intrinsic software redundancy. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 931–942, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2568225.2568287.

[16] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[17] Chun Chen. *Model-Guided Empirical Optimization for Memory Hierarchy*. PhD thesis, USA, 2007. AAI3262764.

[18] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, volume 1, pages 3–9, 1978.

[19] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST-CS98-01, The Hong Kong University of Science and Technology, 1998. `arXiv:2002.12543`.

[20] Ting Chen, Xiao-Song Zhang, Xiao-Li Ji, Cong Zhu, Yang Bai, and Yue Wu. Test generation for embedded executables via concolic execution in a real environment. *IEEE Transactions on Reliability*, 64(1):284–296, 2015. `doi:10.1109/TR.2014.2363153`.

[21] Tsong Yueh Chen, DH Huang, TH Tse, and Zhi Quan Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583. Citeseer, 2004.

[22] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 197–208, 2013.

[23] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000. `doi:10.1145/357766.351266`.

[24] Simon Goldsmith Daniel S. Wilkerson, Scott McPeak. Delta reducer. URL: `https://github.com/dsw/delta`.

[25] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[26] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA):93:1–93:29, 2017. `doi:10.1145/3133917`.

[27] Alastair F. Donaldson, Hugues Evrard, and Paul 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. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13179, doi:10.4230/LIPIcs.ECOOP.2020.22.

[28] Alastair F. Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In *Proceedings of the 1st International Workshop on Metamorphic Testing, MET@ICSE 2016, Austin, Texas, USA, May 16, 2016*, pages 44–47. ACM, 2016. doi:10.1145/2896971.2896978.

[29] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[30] Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. *SIGPLAN Not.*, 47(1):533–544, January 2012. doi:10.1145/2103621.2103719.

[31] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 147–158, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1831708.1831728.

[32] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. doi:10.1145/1064978.1065036.

[33] Wolfgang Grieskamp, Xiao Qu, Xiangjun Wei, Nicolas Kicillof, and Myra B. Cohen. Interaction coverage meets path coverage by smt constraint solving. In Manuel Núñez, Paul Baker, and Mercedes G. Merayo, editors, *Testing of Software and Communication Systems*, pages 97–112, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[34] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: delta debugging, even without bugs. *Software Testing, Verification and Reliability*, 26(1):40–68, 2016. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1574, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1574, doi:https://doi.org/10.1002/stvr.1574.

[35] Jun Gu, Paul W Purdom, John Franco, and Benjamin W Wah. Algorithms for the satisfiability (sat) problem: A survey. Technical report, Cincinnati Univ oh Dept of Electrical and Computer Engineering, 1996.

[36] Ralph Guderlei and Johannes Mayer. Statistical metamorphic testing testing programs with random output by means of statistical hypothesis tests and metamorphic testing. In *Seventh International Conference on Quality Software (QSIC 2007)*, pages 404–409. IEEE, 2007.

[37] Zhan-Wei Hui, Song Huang, Hui Li, Jian-Hao Liu, and Li-Ping Rao. Measurable metrics for qualitative guidelines of metamorphic relation. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 417–422. IEEE, 2015.

[38] Randi Rost John Kessenich, Dave Baldwin. The opengl shading language, 2014. URL: `https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.40.pdf`.

[39] Upulee Kanewala and James M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–10, 2013.

[40] Upulee Kanewala, James M. Bieman, and Asa Ben-Hur. Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels. *Software Testing, Verification and Reliability*, 26(3):245–269, 2016. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1594`, `arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1594`, `doi:https://doi.org/10.1002/stvr.1594`.

[41] Hyondeuk Kim and Fabio Somenzi. Finite instantiations for integer difference logic. In *2006 Formal Methods in Computer Aided Design*, pages 31–38. IEEE, 2006.

[42] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Form. Asp. Comput.*, 27(3):573–609, May 2015. `doi:10.1007/s00165-014-0326-7`.

[43] Andrei Lascu, Matt Windsor, Alastair F. Donaldson, Tobias Grosser, and John Wickerson. Dreaming up metamorphic relations: Experiences from three fuzzer tools. In *6th IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2021, Madrid, Spain, June 2, 2021*, pages 61–68. IEEE, 2021. `doi:10.1109/MET52542.2021.00017`.

[44] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 216–226, New York, NY, USA, 2014. Association for Computing Machinery. `doi:10.1145/2594291.2594334`.

[45] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1, 12 2018. doi:10.1186/s42400-018-0002-y.

[46] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 65–76. ACM, 2015. doi:10.1145/2737924.2737986.

[47] Huai Liu, Xuan Liu, and Tsong Yueh Chen. A new method for constructing metamorphic relations. In *2012 12th International Conference on Quality Software*, pages 59–68, 2012. doi:10.1109/QSIC.2012.10.

[48] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428264.

[49] David Maciver and Alastair F. Donaldson. Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPIcs*, pages 13:1–13:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ECOOP.2020.13.

[50] David R. MacIver. What is hypothesis?, 2007. URL: https://hypothesis.works/articles/what-is-hypothesis/.

[51] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in smt solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 701–712, 2020.

[52] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[53] Scott McMaster and Atif M Memon. Call stack coverage for test suite reduction. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 539–548. IEEE, 2005.

[54] Barton Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33:32–44, 12 1990. doi:10.1145/96267.96279.

[55] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. Random testing of c compilers targeting arithmetic optimization. 2012.

[56] Kazuhiro Nakamura and Nagisa Ishiura. Random testing of c compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 676–679. IEEE, 2016.

[57] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250734.1250746.

[58] John Neystadt. Automated penetration testing with white-box fuzzing, 2009. URL: https://docs.microsoft.com/en-us/previous-versions/software-testing/cc162782(v=msdn.10).

[59] Aina Niemetz and Mathias Preiner. Bitwuzla at the SMT-COMP 2020. *CoRR*, abs/2006.01621, 2020. URL: https://arxiv.org/abs/2006.01621, arXiv:2006.01621.

[60] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. Diffuzz: differential fuzzing for side-channel analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 176–187. IEEE, 2019.

[61] Moritz Pflanzer, Alastair F. Donaldson, and Andrei Lascu. Automatic test case reduction for opencl. In *Proceedings of the 4th International Workshop on OpenCL, IWOCL 2016, Vienna, Austria, April 19-21, 2016*, pages 1:1–1:12. ACM, 2016. doi:10.1145/2909437.2909439.

[62] John Regehr. Reducers are fuzzers, 2015. URL: https://blog.regehr.org/archives/1284.

[63] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 335–346, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254104.

[64] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/73560.73562.

[65] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[66] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[67] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, page 37–48, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1411286.1411292.

[68] Sergio Segura, Gordon Fraser, Ana. B. Sanchez, and Antonio. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016. doi:10.1109/TSE.2016.2532875.

[69] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 263–272, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1081706.1081750.

[70] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012. URL: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker.

[71] Asankhaya Sharma, Shengyi Wang, Andreea Costea, Aquinas Hobor, and Wei-Ngan Chin. Certified reasoning with infinity. In Nikolaj Bjørner and Frank de Boer, editors, *FM 2015: Formal Methods*, pages 496–513, Cham, 2015. Springer International Publishing.

[72] Ryan Stansifer. Presburger's article on integer airthmetic: Remarks and translation. Technical Report TR84-639, Cornell University, Computer Science Department, September 1984. URL: `http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639`.

[73] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, pages 299–302, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[74] Sven Verdoolaege. Integer set coalescing. In *International Workshop on Polyhedral Compilation Techniques, Date: 2015/01/19-2015/01/19, Location: Amsterdam, The Netherlands*, 2015.

[75] Sven Verdoolaege. `isl` manual, 2021. URL: `https://libisl.sourceforge.io/manual.pdf`.

[76] Sven Verdoolaege. Presburger formulas and polyhedral compilation. 2021. URL: `https://libisl.sourceforge.io/tutorial.pdf`.

[77] Elaine J. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 11 1982. `arXiv:https://academic.oup.com/comjnl/article-pdf/25/4/465/1045637/25-4-465.pdf`, `doi:10.1093/comjnl/25.4.465`.

[78] Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating smt solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 718–730, New York, NY, USA, 2020. Association for Computing Machinery. `doi:10.1145/3385412.3385985`.

[79] Chunyan Xia, Yan Zhang, and Zhanwei Hui. Test suite reduction via evolutionary clustering. *IEEE Access*, 9:28111–28121, 2021.

[80] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/1993498.1993532`.

[81] Michal Zalewski. Technical "whitepaper" for afl-fuzz. URL: `https://lcamtuf.coredump.cx/afl/technical_details.txt`.

[82] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002. doi:10.1109/32.988498.

[83] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. Using test case reduction and prioritization to improve symbolic execution. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 160–170, 2014.

[84] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.