Metamorphic Fuzzing of C++ Libraries

Andrei Lascu
Imperial College London
London, United Kingdom,
andrei.lascu10@imperial.ac.uk

Alastair F. Donaldson
Imperial College London
London, United Kingdom,
alastair.donaldson@imperial.ac.uk

Tobias Grosser
University of Edinburgh
Edinburgh, United Kingdom,
tobias.grosser@ed.ac.uk

Torsten Hoefler
ETH Zurich
Zurich, Switzerland,
htor@inf.ethz.ch

***Abstract*—We present a method for automated metamorphic fuzzing of software libraries, implemented as an open-source tool, *MF++*, targeting C++ libraries. Our approach works by automatically synthesising equivalent sequences of calls to a library's API based on a user-provided specification, in a randomized fashion. Equivalent call sequences are then tested using randomized inputs, and result mismatches reveal bugs in the library implementation. This is an instance of metamorphic testing: it avoids the oracle problem because we do not need to know the *expected* results of a set of equivalent call sequences, only that their results should match. Automated test case reduction can then be used to find minimized equivalent call sequences that trigger mismatches, as an aid to debugging. We evaluate *MF++* with respect to four SMT solving libraries and two Presburger arithmetic libraries, leading to the discovery of 21 bugs. We have also successfully used *MF++* and its test case reduction facilities to automatically generate small test cases that exercise source code not covered by the regression test suites of various libraries under test. Unlike most test case generation techniques, the tests we synthesise are equipped with an oracle by construction: the equivalence-based oracle offered by our metamorphic approach. We have submitted patches contributing new coverage-enhancing test cases to the *isl*, *Yices2* and *Z3* projects. The developers of these projects have accepted 21 tests based on our patches so far.**

***Index Terms*—metamorphic testing, fuzzing, test case reduction**

I. Introduction

Rigorous testing techniques are particularly important for software *libraries*: a large number of applications may rely on the correctness of a particular library. A high quality manually-written test suite is essential for any serious library implementation, but it is hard for library developers to anticipate the large and diverse number of ways in which the library's functions might be invoked. Techniques for automating the process of library testing are thus valuable.

Randomized testing—also known as *fuzzing*—can be readily applied to find test cases that exercise library code in intricate ways. Traditional mutation-based fuzzers, such as AFL [1] and libFuzzer [2] are useful for finding example inputs that lead to crashes or undefined behaviours, and automatic test generators such as EvoSuite [3] and Randoop [4] can also be used to generate tests that achieve high coverage of a library's source code. However, the *oracle problem* [5] limits the utility of such techniques for functional testing. Without an oracle, one cannot know whether the library has behaved in a functionally-correct manner on a given random input. This is a barrier to bug-finding, and limits the extent that generated tests are useful for regression testing, even when they increase code coverage.

To overcome this, we investigate the following approach for randomized functional testing of software libraries, which uses *metamorphic testing* [6], [7] as a pseudo-oracle. The library developer identifies a number of *derived operations* that can be implemented using the *primitive operations* of the library. A derived operation might directly mirror a primitive operation offered by the library, or might require some combination of primitive operations. For each derived operation, they provide multiple equivalent implementations, some using the library's primitive operations, some using other derived operations, and some using a mixture of both. A sequence of derived operations can then be automatically expanded to a sequence of primitive operations: each derived operation is replaced by the body of one of its implementations, chosen at random. If that implementation invokes other derived operations, they are randomly expanded in turn. Derived operation implementations can be mutually recursive, so that the expansion process can lead to highly complex fully expanded sequences.

Without an oracle for the library under test, there is no way to determine whether the library behaves appropriately when a single expanded sequence of derived operations is applied to an input. Our approach compensates for the lack of a direct oracle by *cross-checking* the behaviour of multiple expansions: a derived sequence is expanded several times, each expansion is executed on a single input, and the computed results are compared. Assuming the implementations of derived operations are indeed equivalent, a result mismatch between two expansions indicates that there must be a bug in one of the library's primitive operations. Test case reduction can then be used to identify shorter equivalent sequences that still trigger a mismatch, providing a good starting point for debugging.

Our approach is an instance of metamorphic testing [6], [7]: each family of equivalent implementations of a derived operation can be seen as a metamorphic relation. Furthermore, there is scope for randomizing the derived operations used in a test, the manner in which sequences are expanded, and the inputs used for cross-checking. All in all this leads to a novel approach for *metamorphic fuzzing* of software libraries.

In addition to bug-finding, our metamorphic fuzzing approach can be used for coverage-guided test case generation. This involves (a) identifying statements that are not covered by a library's test suite but *are* covered during metamorphic fuzzing, (b) injecting failing assertions before such statements,

(c) applying metamorphic fuzzing again to find test cases that trigger the assertion failures (i.e. achieve the additional coverage), and (d) using test case reduction to produce small test cases that still trigger the assertions (i.e. still achieve the extra coverage). Because the resulting reduced tests have oracles *by construction*—they check equivalence of a pair of minimised sequences of operations on a minimised input—they are suitable for adding to library regression test suites.

We have implemented our approach in a tool, *MF++*, intended for use by C++ library developers. Using *MF++* requires some manual effort from the developer. Principally, they identify suitable derived operations and provide equivalent implementations of these (i.e., provide the metamorphic relations), and provide a function that checks whether the results of two operation sequences are equivalent. The developer provides this information in C++, allowing them to work in the same language used for library development, with its powerful features. This manual effort is a *one-off cost*: once these ingredients have been provided, *MF++* is able to produce an *endless* stream of metamorphic test cases that can be used to intensively test the library. If the developer makes mistakes when providing these ingredients (e.g., they provide derived operation implementations that turn out to be inequivalent), *MF++* will quickly flag these up as possible bugs and automated test case reduction can then be used to yield a simple reproducer test case that the developer can examine to determine which operation implementation they need to fix.

We have evaluated *MF++* with respect to 6 libraries: *Z3* [8], *CVC4* [9], *Yices2* [10], and *Boolector* [11] (mature SMT solvers supporting various theories); and *isl* (Integer Set Library) [12] and *Omega* [13] (Presburger arithmetic [14] libraries). Our testing has revealed 21 bugs overall, in *Z3*, *Yices2*, *isl* and *Omega*, 19 of which have been fixed in response to our reports. We have used coverage-guided test case generation to synthesise new, oracle-equipped test cases for the regression test suites of *isl*, *Yices2* and *Z3*, leading to 21 new tests being integrated into open-source test suites so far. These coverage-guided tests have shown value already by identifying two use-after-free bugs in the *Z3* code base.

In summary, our main contributions are:

1) A novel approach to metamorphic fuzzing of libraries that allows randomized testing for *functional* properties;
2) The design and implementation of *MF++*, an implementation of this approach for C++ libraries;
3) The combination of metamorphic fuzzing and test case reduction to enable coverage-guided test case generation;
4) A large evaluation over 6 libraries, leading to the discovery and fixing of many previously unknown bugs;
5) A demonstration of the effectiveness of coverage-guided test case generation, with 21 new test cases already integrated into open-source library test suites.

Reproducibility. *MF++* is open source [15], and we have prepared an artifact that allows experimenting with the version of the tool used for the experiments presented in this paper [16].

II. Background

Metamorphic testing. Metamorphic testing aims to overcome the oracle problem [5] by exploiting expectations on how the outputs of a system under test (SUT) should be related when applied to inputs that are associated in specific ways. As an example (adapted from [7]), suppose program P computes the length of the shortest path between two points in an undirected graph. For two nodes a and b in a graph G, we can immediately tell that P is faulty if we find that $P(G, a, b) \neq P(G, b, a)$. Importantly, we can know that P is faulty without requiring an oracle for P—i.e., without actually knowing the length of the shortest path from a to b.

The key to metamorphic testing is that, from domain knowledge, some *metamorphic relations* (MRs) are expected to hold for the SUT. Formally, an MR is a pair of binary relations (R, S), such that if $(x, y) \in R$ for inputs x and y, then $(f(x), f(y)) \in S$, where $f(a)$ results from executing the SUT on input a. In our example, $R = \{((G, a, b), (G, b, a)) \mid a \text{ and } b \text{ are nodes of undirected graph } G\}$, and S is "=". Metamorphic testing involves checking whether an MR indeed holds for a selection of pairs of inputs. When an MR is violated this either indicates that there is a bug in the SUT (if the MR is accurate), or that the MR was poorly formulated.

Test case reduction. Randomized testing can lead to bug-triggering test cases that are too large and complex for developers to understand. A test case reducer takes a large bug-triggering test case and yields a smaller, simpler test case that still triggers the bug. Most test case reducers are based on delta debugging [17], which involves systematically attempting to remove portions of the test case that may be unnecessary for triggering the bug of interest. Hierarchical delta debugging [18] allows more efficient reduction by exploiting domain-specific information about the structure of test cases. When a test case triggers a functional error, it is important to preserve *validity* of the test case during the reduction process. In §III-C and §IV-C we describe the design and implementation of a customised validity-preserving reducer tailored to work with our metamorphic fuzzing approach.

III. Details of Our Approach

Our approach to metamorphic library fuzzing repeatedly:

- generates multiple equivalent sequences of calls to functions of the library under test;
- executes each equivalent sequence on the same randomly-generated input;
- checks that the results computed by the sequences are all equivalent;
- in the event of a mismatch, applies test case reduction to yield minimised equivalent call sequences that expose the mismatch, to aid in debugging.

It is the use of *equivalent* sequences that makes this a metamorphic testing approach. These equivalences can be viewed as metamorphic relations, and allow bugs to be identified without knowledge of the result that a particular sequence should compute, i.e., without an explicit oracle.

We now describe the approach in detail, focusing on the ingredients the user needs to provide (§III-A), and how these ingredients are used for metamorphic fuzzing (§III-B) and exploited during test case reduction (§III-C). We then explain an alternative use case for our approach: coverage-guided generation of oracle-equipped test cases (§III-D). We also discuss the rationale for requiring the user to specify equivalent implementations of derived operations directly, rather than attempting to synthesise them automatically (§III-E).

Running example. Suppose we wish to test an arbitrary-precision integer arithmetic library that defines: a type, `BigInt`, representing mathematical integers; binary functions `add`, `sub` and `mul` and `greaterThan` implementing the $+$, $-$, $\times$ and $>$ operations; a unary function `neg` implementing negation. We refer to these functions as the *primitive operations* of the library. Suppose that literals are overloaded so that an `int` literal can be used in a `BigInt` context.

A. User-provided Ingredients

To use our approach, the user must provide a number of ingredients upfront. The payback for this one-off cost is the ability to subsequently generate an unbounded number of randomized metamorphic tests, with further manual effort required only if the library's API changes, or if the user wishes to expose features of the library in a more detailed fashion.

Derived operations. The user must first decide on some *derived operations* that can be implemented using the library under test. These might directly mirror primitive operations of the library, or might be operations that can only be implemented by combining primitive operations. For our example, we consider five derived operations: `ADD` and `MUL`, which mirror the `add` and `mul` primitive operations, `IDENTITY`, which takes a `BigInt` and should return a `BigInt` with the same value, `ABS`, which should yield the absolute value of a `BigInt`, and `ZERO`, which should yield a `BigInt` with the value 0.

Equivalent implementations of derived operations. For each derived operation the user must provide at least two *equivalent implementations*. Each operation should have at least one *base case* implementation that uses only primitive operations of the library; e.g., a base case implementation of `MUL` would delegate to the `mul` primitive operation, while a base case implementation of `ABS` could be:

```
BigInt ABS_BASIC(BigInt x) {
  return greaterThan(x, 0) ? x : neg(x); }
```

There should also be at least one implementation for each derived operation that achieves the effect of the operation in a more complex way, using a combination of primitive and derived operations; this is to ensure that the space of possible equivalent implementations of derived operations is large and interesting. A more interesting implementation of `MUL` could implement multiplication by repeated addition:

```
BigInt MUL_BY_ADD(BigInt x, BigInt y) {
  BigInt result = ZERO();
  for (BigInt i = ZERO(); greaterThan(y, i);
       i = ADD(i, 1))
    result = ADD(result, x);
  return result;
}
```

Instead of using the `add` function directly, we use the derived operation `ADD`; similarly the literal 0 is replaced by `ZERO`.

For `ADD`, we could consider various implementations in addition to the basic implementation that delegates to `add`, e.g., exploiting the commutative law of addition:

```
BigInt ADD_COMMUTED(BigInt x, BigInt y) {
  return add(y, x); }
```

or expressing addition in terms of subtraction and negation:

```
BigInt ADD_BY_SUB(BigInt x, BigInt y) {
  return sub(a, neg(b)); }
```

The basic `IDENTITY` implementation would directly return its argument. A more complex implementation could be:

```
BigInt IDENTITY_ADD_ZERO(BigInt x) {
  return ADD(x, ZERO());  }
```

Notice that this does not refer to any primitive operations of the library directly, but only to other derived operations.

An example alternative implementation of `ABS` is:

```
BigInt ABS_BY_SUB_AND_NEGATE(BigInt x) {
  return greaterThan(x, ZERO())
      ? x : sub(x, MUL(2, neg(x))); }
```

Finally, suppose function `rand` returns a random `BigInt`. We can complement the base case of `ZERO`, e.g:

```
BigInt ZERO_BY_SUB() {
  BigInt temp = rand();
  return sub(IDENTITY(temp), IDENTITY(temp)); }

BigInt ZERO_BY_MUL() {
  return MUL(rand(), ZERO()); }
```

Random generation of library inputs. The user must provide a means of randomly generating inputs of each data type that a derived operation can consume. In §IV we describe how the *MF++* tool facilitates this. In our example, this would require a source of random `BigInt` values, which could be achieved by generating random machine integers and promoting them to `BigInt`. The level of sophistication associated with random generation is at the discretion of the user: basic random generation suffices to get started with our technique, but more sophisticated generation to ensure particular properties or distributions over input values might improve the effectiveness of testing.

Equivalence checks. The user provides a function, $\mathsf{equivalent}(x_1, x_2)$, that decides whether two results generated by the library under test are equivalent. The required notion of equivalence for a particular library is usually obvious. We give some practical examples when we discuss our case studies in §V. For our `BigInt` example, equivalent means equal.

B. Randomized Metamorphic Testing

We now explain how the above user-provided ingredients are used for metamorphic fuzzing.

Expanding derived operations. A derived operation is *expanded* by selecting one of its implementations and recursively expanding any derived operations used therein, until no derived

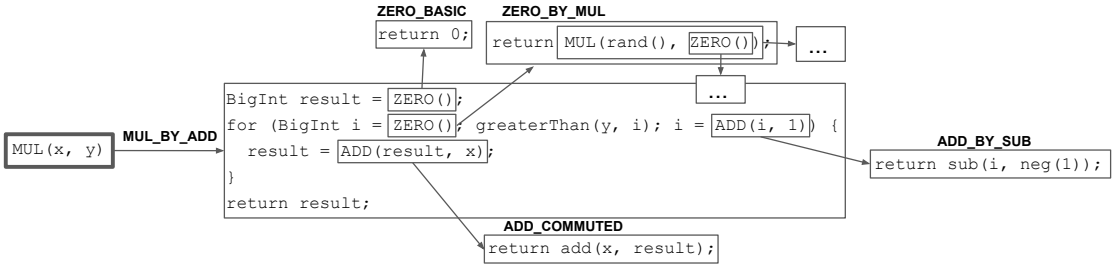

Fig. 1: Recursively expanding the `MUL` derived operation

operations remain. Figure 1 shows part of a possible expansion of the `MUL` operation of our running example. First `MUL` is expanded to `MUL_BY_ADD`, which involves two occurrences each of the `ADD` and `ZERO` derived operations. The instances of `ADD` are expanded to `ADD_COMMUTED` and `ADD_BY_SUB`, which do not use further derived operations, so their expansion is complete. One `ZERO` instance is expanded to `ZERO_BASIC`, the other to `ZERO_BY_MUL` which uses the `MUL` and `ZERO` derived operations; these would in turn need to be expanded.

Let $\mathsf{expansions}(OP)$ denote the (typically infinite) set of possible expansions of derived operation OP. For an expansion $E \in OP$, let $\mathsf{execute}(E, \vec{x})$ denote the result obtained by executing E on input $\vec{x}$ using the library under test. We have the following *metamorphic relation* (MR): if $E_1, E_2 \in \mathsf{expansions}(OP)$ and $\vec{x}$ is a vector of inputs, then $\mathsf{equivalent}(\mathsf{execute}(E_1, \vec{x}), \mathsf{execute}(E_2, \vec{x}))$ should hold. Using the formal definition of an MR in terms of binary relations R and S (§II) we have $R = \{((E_1, \vec{x}), (E_2, \vec{x})) \mid \exists OP \,.\, E_1, E_2 \in \mathsf{expansions}(OP)\}$, and $S = \mathsf{equivalent}$.

The testing process. A stream of random metamorphic test cases is generated by repeating the following process.

A sequence of m initial input variables, $x_1, \ldots, x_m$, are declared, each initialized to a randomized value. A sequence of n derived operations $OP_1, OP_2, \ldots, OP_n$ is then selected at random. A particular derived operation may appear multiple times in this sequence. For each $1 \leq i \leq n$, a temporary variable t_i matching the return type of OP_i is introduced to capture the value returned by OP_i.[1] For each argument required by OP_i, a random choice is made between the in-scope variables, i.e., the input variables $x_1, \ldots, x_m$ and the result variables $t_1, \ldots, t_{i-1}$ of earlier operations.

This leads to a set of input variables $x_1, \ldots, x_m$ (for some $m \geq 0$), each with a randomized initial value, and a sequence of n derived operation invocations. The ith invocation in the sequence has the form $t_i = OP_i(\vec{p_i})$, where each parameter in $\vec{p_i}$ is either one of the input variables x_j $(1 \leq j \leq m)$, or the result t_j $(1 \leq j < i)$ of an earlier operation.

The invocation sequence is then replaced with k *expanded* sequences, called *equivalent variants*. A variant v $(1 \leq v \leq k)$ is obtained by duplicating the invocation sequence, and then:

- replacing each definition and use of a result variable t_i with a variant-specific result variable $t_{i,v}$;
- replacing each derived operation OP_i with a variant-specific expansion $E_{i,v} \in \mathsf{expansions}(OP_i)$.

The final step in generating a metamorphic test involves asserting a *postcondition*: that $\mathsf{equivalent}(t_{n,1}, t_{n,v})$ holds for each $2 \leq v \leq k$. Because each variant v is expanded from the same sequence of derived operations, any one of these assertions failing demonstrates that the equivalent expansions of the sequence of derived operations have led to different results. Such an assertion failure indicates that at least one library function must be implemented *incorrectly*. The failure can be investigated to pinpoint the bug in the library implementation.

C. Test Case Reduction

A test that triggers a postcondition failure may be very complex and hard to understand: the sequence length n and number of variants k may be large, certain derived operations may have been expanded with a high recursion depth, and the random inputs used for testing may also be intricate.

We propose a specialised form of hierarchical delta debugging [18] to shrink tests down to a more digestible form. First, variants can be systematically eliminated to identify the subset of variants required to trigger the postcondition failure. Derived operations can then be removed from the sequences of the remaining equivalent variants, leading to a minimal set of derived operations. The expansions of the derived operations that remain can then be systematically simplified: each time a non-base case implementation of a derived operation has been used, the base case implementation can be tried instead, so that non-trivial implementations remain only where they are required in order for the postcondition failure to occur. Finally, the test input that induces the failure can be simplified.

We note again that test case reduction is useful not only for diagnosing bugs, but also for helping the developer to quickly identify bugs in the ingredients they have provided.

D. Coverage-guided Test Case Generation

As discussed in the introduction, an alternative application of our approach, beyond bug finding, is generation of oracle-equipped test cases that increase coverage of the library under test. We focus on statement coverage, but the approach could be applied in the context of other measurable coverage criteria.

[1]For ease of presentation we assume operations have non-void return type; in practice result variables are omitted for operations with void return type.

Suppose a library's regression test suite covers a set S of statements of the library implementation. If a randomly-generated test case t covers a statement $x \notin S$, then t identifies a *coverage gap*: x is *not* covered by the current regression test suite, but coverage of x is achievable, as demonstrated by the test case t. The test case t has the *potential* to be used to augment the library's regression test suite. However, being randomly-generated, t is likely to be large and complex. Furthermore, if t does not come equipped with a test oracle, there is little to be gained by adding t to the regression test suite since the pass/fail status of t cannot be determined.

Our metamorphic fuzzing approach provides a solution to these problems. First, generated test cases come with an equivalence-based oracle by construction: a test case checks that a number of equivalent variants do indeed compute equivalent results. Second, test case reduction (§III-C) allows shrinking a large test case to a small one that is *still* equipped with a metamorphic oracle. This a form of *cause reduction* [19], [20]: instead of reducing a test case with respect to a failure, reduction is performed with respect to a coverage target.

Our coverage-guided approach differs fundamentally from feedback-directed techniques such as EvoSuite [3] and Randoop [4]. These techniques exploit feedback from the SUT during test generation. In contrast, *MF++* is a black-box fuzzer. However, when *MF++* turns out to have covered additional code compared to a regression test suite, the process of using test case reduction to obtain a small test is coverage-guided.

We explain how we have implemented this idea in §IV-D, and discuss our experience using this approach to contribute new test cases to open source projects in §V-C.

E. Discussion

Our approach requires the library developer to identify suitably-interesting derived operations, and provide implementations that are indeed equivalent. The usefulness of the test cases that can then be generated by our approach—in terms of their bug-finding ability—is intimately related to the derived operations (and implementations thereof) that the developer has provided: trivially-equivalent implementations are unlikely to lead to the discovery of bugs.

As emphasised above, the provision of derived operation implementations by the user is a one-off effort, and test case reduction serves as a debugging aid when the user gets things wrong. One might ask whether equivalent implementations of derived operations could be synthesised *automatically*, to remove this burden completely. We regard this as an interesting direction for future research, but believe that it would be very challenging in the context of C++ libraries. Furthermore, a library developer would need to understand the details of a synthesised implementation in order to comprehend bugs found by the approach, and their time might be better spent using their domain expertise to write some simple implementations of derived operations directly.

IV. Design and Implementation of *MF++*

The approach described in §III is general and could be instantiated in principle for the testing of libraries written in any programming language. We now describe the implementation of the *MF++* tool for testing C++ libraries (and also C libraries, due to the ease by which C code can be invoked from C++).

```
#include "bigint.h" // Include library header
#include "mfpp.h"   // Include MF++ header

namespace mfpp {

namespace operations {
  // Forward-declare signatures for derived operations
  namespace ADD { BigInt placeholder(BigInt, BigInt); }
  namespace MUL { BigInt placeholder(BigInt, BigInt); }
  namespace ZERO { BigInt placeholder(); }
  // Similar for other derived operations

  namespace ADD { // Equivalent implementations of ADD
    BigInt ADD_BASIC(BigInt a, BigInt b) {
      return add(a, b);
    }
    BigInt ADD_BY_SUB(BigInt a, BigInt b) {
      return sub (a, neg (b));
    }
    // Other implementations of ADD
  }

  namespace MUL { // Equivalent implementations of MUL
    BigInt MUL_BASIC(BigInt a, BigInt b) {
      return mul(a, b);
    }
    BigInt MUL_BY_ADD(BigInt a, BigInt b) {
      BigInt result = ZERO::placeholder();
      for (BigInt i = ZERO::placeholder();
            greaterThan(b, i);
            i = ADD::placeholder(i, 1))
        result = ADD::placeholder(result, a);
      return result;
    }
    // Other implementations of MUL
  }

  // More namespaces with implementations for ZERO, etc.
} // namespace operations

namespace generators {
  BigInt rand() { return BigInt(std::rand()); }
  BigInt byAdd(BigInt a, BigInt b) { return add(a, b); }
  // Similar for other operators.
} // namespace generators

namespace checks {
  bool equivalent(BigInt a, BigInt b) {
    return a == b;
  }
} // namespace checks
} // namespace mfpp

int main() {
  initializeBigInt(); // Any necessary initialization
  BigInt x1 = 42;
  BigInt x2 = mfpp::fuzz<BigInt>();
  // ...
  BigInt xm = mfpp::fuzz<BigInt>();
  mfpp::meta_test();
  finalizeBigInt(); // Any necessary finalization
  return 0;
}
```

Fig. 2: *MF++* specification for the `BigInt` example

A. Specifying a Library

To use *MF++* a developer writes a C++ file that (a) includes the library header files plus an *MF++*-specific header, (b) uses a series of C++ namespaces to describe the derived operations and other user-provided ingredients, and (c) declares a `main` method that serves as a template for the form of generated

tests. Collectively we call this a *specification* for the library under test. Figure 2 illustrates a specification for the `BigInt` example of §III. We discuss its various components.

Derived operations and implementations. An `operations` namespace (lines 6–39 of Figure 2) is used to expose derived operations and their implementations to *MF++*. Each operation has its own sub-namespace, so there are namespaces for `ADD`, `MUL`, etc., in our example. First, for each derived operation, a function called `placeholder` is forward-declared. This specifies the signature of the derived operation. The forward declaration for the signature of `ADD` is on line 8, for example.

The forward declaration of a derived operation is followed by its equivalent implementations. Two implementations of `ADD` are shown on lines 13–21, inside the `operations::ADD` namespace, matching the signature of `ADD::placeholder`. Two implementations of `MUL` are shown on lines 23—36. Recall from §III that `MUL_BY_ADD` uses the derived operations `ZERO` and `ADD`. In the implementation of Figure 2 this is codified via calls to `ZERO::placeholder` (lines 28 and 29) and `ADD::placeholder` (lines 31 and 32). Because an *MF++* specification is written in C++, implementations of derived operations can use the full power of this language.

No implementations are provided for the `placeholder` functions; they are literally placeholders. Calling e.g., `ADD::placeholder` indicates that *some* implementation of `ADD` should be invoked.

Random input generation. Our approach requires a source of random initial inputs, and it can be useful for implementations of derived operations to have access to random generation (see the `ZERO_BY_SUB` and `ZERO_BY_MUL` examples of §III). *MF++* provides a templated function, `mfpp::fuzz<T>()`, that can be invoked to request a randomly-generated value of type `T`. The ingredients that `fuzz` can use are specified via a `generators` namespace, illustrated on lines 41–45 of Figure 2. This allows *MF++* to expand a call to `mfpp::fuzz<BigInt>()` to call either `generators::rand` or `generators::byAdd`. In the latter case, *MF++* would need to provide two further `BigInt` values, one for each parameter of `generators::byAdd`, which it could do by using `fuzz` to generate additional values, or re-using previously-generated values. The `fuzz` function can also be invoked from implementations of derived operations when random data is required. By exposing a number of constructor operations to *MF++*, the user equips *MF++* with the ability to recursively generate interesting inputs.

Equivalence checks. A `checks` namespace, illustrated by lines 47—51 of Figure 2, must include at least one binary predicate that checks whether two values of one of the library's data types are equivalent. More than one such function can be specified, so that if multiple checks are desired in the postcondition of a test they can be specified separately. *MF++* will ensure that the return type of the final operation in the operation sequence that it generates matches the argument types of the functions in the `checks` namespace (the argument types must be the same for all such functions).

The test template. A `main` method specifies the skeleton structure of a generated test. This should perform library setup (line 55 of Figure 2), declare a number of input variables (lines 56– 59), use a special `mfpp::meta_test` function to instruct *MF++* as to where it should insert the equivalent operation sequences and postcondition checks (line 60), and perform library tear-down (line 61). The harness can use a mixture of randomly- and concretely-initialized inputs. In our example, `x1` is initialized with a specific value (line 56), while `x2` and `xm` are initialized to random values via calls to `mfpp::fuzz<BigInt>()` (lines 57 and 59).

B. Implementing Test Case Generation

Given a specification, *MF++* generates a test by outputting a fresh C++ file that `#includes` the library's headers, and then contains a copy of the `main` function adapted such that (1) each `mfpp:fuzz<T>()` call is expanded to a concrete series of calls to functions from the `generators` namespace; and (2) `mfpp:meta_test()` is replaced with a series of equivalent expansions of a randomly sequence of derived operations, followed by a post-condition check, according to the procedure described in §III. All variables in scope at the `mfpp:meta_test()` call site are available as arguments to the derived operation sequence.

MF++ is implemented using the Clang LibTooling framework [21], which provides code analysis facilities for C++.

C. Implementing Test Case Reduction

We have implemented the customised version of hierarchical delta debugging discussed in §III-C, again using the Clang LibTooling framework. The reducer regards a test case as *interesting* if the test case exhibits a postcondition failure—i.e., there is a result mismatch between metamorphic variants—or causes the library implementation to crash in some other way (e.g., due to an internal error being triggered).

The reducer exploits domain-specific knowledge of the format of generated test cases in order to simplify them. It first systematically eliminates metamorphic variants that are unnecessary for the bug to trigger. This usually eliminates all but two variants. The reducer then systematically shortens the sequence of derived operations executed by the remaining variants. Eliminating the ith derived operation involves removing the respective expansions of this operation from the remaining variants. If the result of the ith derived operation is used later in the test case, such uses are replaced with uses of other type-compatible variables. The reducer then (a) folds up non-trivial expansions of derived operations by attempting to replace them with base cases, and (b) simplifies the input to the test case. In both cases, the reducer exploits the format of the library specification to guide the simplification process.

Each of these phases iterates until a fixed-point is reached—i.e., to the point where no further simplifications of the kind associated with the phase lead to an interesting test case. All phases start by aggressively trying many simplifications simultaneously, in order to quickly cut out large swathes of

a test case where possible, and gradually reduce the level of aggression until simplifications are attempted one at a time.

In §V-D we provide experimental data showing the effectiveness of test case reduction in terms of the extent to which test cases can be reduced, and the time the process takes.

D. Coverage-guided Test Case Generation in Practice

To realise the coverage-guided test case generation approach of §III-D, we compile a library under test in debug mode (to disable optimizations), enabling `gcov`-based coverage instrumentation [22]. We execute the library's regression test suite, capturing the associated coverage data. Separately, we conduct a fuzzing run against the library using *MF++* for several hours and capture the associated coverage data.

We used scripts provided by the GraphicsFuzz project [23] to perform differential analysis of the coverage data. This allows us to identify statements covered by *MF++* but not by the regression test suite. We temporarily edit the library source code to insert a failing assertion before each novel coverage point of interest. This means that hitting the coverage point will appear to cause the library to crash.

We rebuild the modified library (without coverage instrumentation), and conduct another fuzzing run using *MF++*. This time, each novel coverage point will cause an assertion to fail. *MF++* will find test cases that trigger the assertion failures, and reduce them to minimal examples that still trigger the assertion failures—i.e., that still yield the additional coverage.

As discussed in §III-D the reduced test cases remain equipped with a metamorphic oracle, so that they are suitable for contributing back to the library's regression test suite, after appropriate manual clean-up to make them more human readable. This process is fully automatic, except for the decision as to which coverage points to instrument, and the manual cleanup.

We describe our positive experience using this process to contribute test cases to various library test suites in §V-C.

V. Evaluation

We evaluate *MF++* over 6 software libraries: the SMT solvers *Z3* [8], *CVC4* [9], *Yices2* [10] and *Boolector* [11], and the libraries for Presburger arithmetic *isl* [12] and *Omega* [13]. We justify our choice of these libraries and provide details of the *MF++* specifications for them in §V-A.

Our evaluation then focuses on three aspects: the ability of *MF++* to find previously-unknown bugs (§V-B), the ability of *MF++* to synthesise tests that achieve new coverage and the extent to which library developers are receptive to such test cases (§V-C), and the throughput of *MF++* and efficiency of its test case reducer (§V-D).

In principle it would be interesting to compare *MF++* with other randomized test case generation tools, such as Randoop [4] or EvoSuite [3]. Aside from the practical difficulty presented by these tools being geared towards a different programming language (Java), the lack of a general oracle for generated tests would make such a comparison difficult. The metamorphic nature of *MF++* means that generated tests are equipped with oracles by construction, based on user-provided specifications. This is not true for more fully-automated test case generation techniques. As a result, any comparison would have to be limited to e.g., comparing the degree of code coverage achieved by generated tests, or the extent to which generated tests can trigger bugs according to a trivial oracle (e.g., the library crashing) is available. Even then, a comparison would be complicated by the fact that it is often acceptable for a library to crash when invoked in an invalid manner. Because *MF++* is guided by user-provided specifications, the test cases it generates invoke the library in legitimate ways. This is not true for test cases generated by more automatic methods.

A. Libraries Under Test

There are two main motivations for the choice of libraries. First, due to the nature of *MF++*, namely that domain knowledge of a library under test is greatly desired when writing specifications, three libraries were chosen due to familiarity: *Z3*, *CVC4* and *isl*. Then, we evaluated *MF++* over three similar libraries, *Yices2*, *Boolector* and *Omega*, to understand how well our design choices apply to translating a specification in the same domain, but a distinct API. For *Omega*, we found two bugs that triggered very frequently. We reported these to the library maintainers and were advised that *Omega* is no longer supported, thus we did not test it further.

For the SMT solvers, we designed two conceptual specifications: one over the theory of quantifier-free non-linear integer arithmetic, the other over the quantifier-free theory of bit-vectors. We coded these up as concrete *MF++* specifications for the SUTs. We refer to these as the *integer* and *bit-vector* specifications henceforth. Most of our derived operations map to existing operations over the respective sort (e.g., addition or multiplication for integers, addition or xor for bit-vectors). We added additional derived operations, e.g., `ABS` for bit-vectors.

For each SMT specification we perform the following checks over a pair of metamorphic variants (x_1, x_v). First, we assert that $x_1 \neq x_v$ is not SAT: since the formulas are equivalent by construction, a violation of this property would constitute a solver bug. (We do not assert that $x_1 \neq x_v$ is UNSAT, because the solver may legitimately return UNKNOWN if the theory is not decidable.) We then attempt to test the model generation capabilities of the solver. We check whether $x_1 == 0$ is satisfiable. If it is, we ask the solver for a model, and we assert that the resulting model is also a model for $x_v == 0$: because x_1 and x_v are equivalent, $x_1 == 0$ and $x_v == 0$ are also equivalent and thus a model for one (if it exists) must hold for the other. If $x_1 == 0$ is unsatisfiable, then we check that $x_v == 0$ is *not* satisfiable.

We tested *Z3* and *CVC4* with both the integer and bit-vector specifications. *Boolector* does not support integer theories so we tested it only with the bit-vector specification, and (due to time limitations) we restricted *Yices2* to this specification due to the similarity between it and *Boolector*'s APIs.

For *isl*, we used a specification that generates sets in an n-dimensional space by randomly selecting points in the space, creating a singleton set for each point, then uniting all of these

sets together, and finally computing the convex hull of the resulting set. We found that the generation phase was the most interesting part of an *isl* test case, as we could more finely tune what sort of inputs *isl* should consume, and subsequent operations over these specific inputs would more likely expose issues. For the derived operations, most of them take inspiration from Boolean algebra, such as DeMorgan's laws. The check uses the internal `is_equal` function to check for set equality.

B. Bugs found using MF++

A key measure of the usefulness of a randomized test generator is its ability to find bugs in practice. Because *MF++* is driven by a user-provided specification, its usefulness is a function of both the quality of a given specification, and the mechanism the tool uses to generate tests. During the development of *MF++*, we have found and reported 21 bugs across four libraries: 5 in *Z3*, 11 in *isl*, 2 in *Yices2*, and 3 in *Omega*. Except the 3 *Omega* bugs, all bugs have been fixed in response to our reports. Out of all 21, we classify 10 as functional bugs, requiring metamorphic checks to be triggered. The identification of these bugs provides evidence that *MF++*, paired with the library specifications that we have implemented, has practical value. We now discuss a selection of these bugs.

***Z3*.** We describe two example bugs. The first bug (discovered at API level) is illustrated by this SMTLIB formula:

```
(assert (= x -2))
(assert (= y (- -2 (div (* -2 x) -2))))
(assert (not (= y 0)))
```

It is easy to see that the formula is UNSAT, as the value of y is constrained to be 0, but *Z3* incorrectly yielded the result SAT. Once reported [24], the issue was promptly fixed. The maintainer pointed us to a separate issue [25] opened just 9 days prior to our report, for which the deployed fix contained the bug identified via our testing mechanism. The short time it took for us to identify a regression bug shows how the technique can be used to augment regression testing.

A second bug [26] had to do with the commutativity of the $\neq$ operator. Two equivalent by construction formulas would make Z3 report $x \neq y$ as SAT, but $y \neq x$ as UNSAT.

A third bug [27] involved the formula `assert (<= 0 (^ 2 -1)))` yielding UNKNOWN, when it is trivially satisfiable via constant propagation.

***isl*.** Our testing led to the fixing of a complex issue in *isl*'s *coalesce* routine [28]. Three separate *MF++* bug reports led the *isl* developers to the problem. We discuss the simplified test case committed in the first patch of this bug-series. Namely, consider an integer set declared by the two following disjuncts:

$$0 \leq x, y, z \leq 100 \wedge 0 < z \leq 2 + 2x + 2y$$

$$z = 0 \wedge x, y \leq 100 \wedge y \leq 9 + 11x \wedge x \leq 9 + 11y$$

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 $x, y \geq 0$. Since the constraint is not redundant for the first disjunct entirely, it means it is redundant (with respect to $x, y \geq 0$) on the hyperplane $z = 0$. Thus, we assume the constraint is valid for integer points.

The three bugs we found all apply to a specific component of the coalesce routine. In this component *isl* searches for pairs of adjacent polyhedra which can be combined by rotating a constraint of the first polyhedron until the enlarged first polyhedron includes all integer points of the second. While coalescing may result in additional rational points in the expanded polyhedron, for correctness it is essential that the number of integer points remains unchanged. The first bug [29], 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. Our second test case then exposed a problem where some newly redundant constraints were ignored for polyhedrons that did not yet exist before the coalescing routine was called but were only created as part of the iterative coalescing process itself. While the second patch addressed this instance of incorrectly updated state, our third test case showed that the coalescing routine still relied on inconsistent state. The final solution implemented by the *isl* developers removed the earlier generalisation.

***Yices2*.** The *Yices2* API allows the `xor` operation to be applied via `yices_bvxor` for an arbitrary number of operands, or `yices_bvxor2` and `yices_bvxor3` for two and three operands, respectively. As we included the three operand version in our specification, we discovered a bug whereby the *Yices2* API mistakenly called `yices_bvor` instead of `yices_bvxor` in the implementation of `yices_bvxor3`. The bug was promptly fixed following our report [30].

C. Using MF++ for Coverage-guided Test Case Generation

Through some pilot experiments, we found that *MF++* was able to cover a substantial number of statements in the *Z3*, *Yices2* and *isl* code bases not covered by their regression test suites, so we decided to focus our coverage-guided test case generation efforts on these libraries. We liaised with the library developers to check whether they would be receptive to tests that work at the API level, which aim to increase code coverage rather than to expose known faults. Developers from all three projects were receptive to the idea. A *Yices2* developer commented *"We are absolutely interested in integrating your tests in Yices."*, *"API tests would be especially useful"*, and *"If you could focus on API and not convert [to] SMT2 that would be more interesting to us"*; a *Z3* developer commented: *"Tests added to the examples/c++ directory could be very useful and welcome"* (this is the directory for *Z3*'s API tests). So far we have proposed 21 new test cases to these libraries: 10 to *Z3*, 10 to *Yices2* and 1 to *isl*. We only proposed a single *isl* test because the *isl* test suite is written in C and *MF++* interfaces with the *isl* C++ API to yield C++ tests; we translated one test to C as a proof of concept. Our contributions to *Z3* and *Yices2* were directly accepted; the projects even made infrastructural changes to accommodate our contributions as a new category of API tests. The lead developer of *isl* indirectly integrated our contribution by writing a fresh test case inspired by our test.

TABLE I: Throughput of randomized testing using *MF++*

Library	Test/hour	% generation	% compilation	% execution
Boolector	605.64	36.54	20.77	42.66
CVC4	88.99	16.22	4.75	78.98
isl	256.46	67.49	30.84	1.66
Yices2	711.14	43.87	20.72	35.37
Z3	161.96	36.62	8.15	55.19

The aim of coverage-guided test case generation is to produce small, high quality tests which target a specific, previously untested coverage point. Because they come equipped with a test oracle, the idea is that these tests have future value for regression testing. Our contributions to *Z3* have already shown value in identifying two heap use-after-free errors [31], [32].

The tests we have added so far improve coverage of various components. In *Z3* this includes the non-linear SAT solver, handling of polynomials, and various internal solver edge-cases; in *Yices2* our contributions improve coverage of the C/C++ API, features of model creation and application, and term-handling. Our contributions so far are merely a proof of concept: the additional coverage they provide is small compared to that already achieved by the overall test suite (an improvement of less than 1%). However, the process we have put in place can be further iterated to yield oracle-equipped tests whenever *MF++* is able to cover more code than the regression test suite.

Before submitting tests, we perform further manual reduction than what the reducer automatically provides. In addition to removal of boiler-plate code and simplification of variable names (which could be automated), this process was primarily driven by library-specific semantics that the reducer is unaware of. For example, in *isl*, the objects we generate are created via sequences of API calls. However, we can simplify the generation of a set, for example, by instead creating the set from a string representation, rather than the sequence of API calls, essentially folding multiple API calls to a single function.

D. Performance of Test Case Generation and Reduction

We present some data demonstrating the throughput of testing using *MF++*, and the performance of test case reduction. While there are no suitable existing tools with which to present a performance comparison, we believe this data will be valuable to researchers investigating similar techniques in the future.

To indicate the throughput of testing, Table I shows the average number of test executions per hour for 20-hour runs of each of *Z3*, *isl*, *Yices2*, and *CVC4*, and a 6-hour run of *Boolector*, and the percentage of time spent on test case generation, compilation of test programs, and test execution. These tests were run on a Ubuntu 20.04 Docker container, hosted on a machine with an Intel Core i7-6700 3.40 GHz CPU and 16GB of RAM.

The percentages across each SUT do not add up to exactly 100% due to the system overhead. We observe a rather wide throughput, from 89 tests per hour, up to 711. The internal difference between experimental runs are internal parameters (e.g., number of variants, recursion depth), and a few SUT-specific functions that we defined based on specific capabilities.

TABLE II: Performance of test case reduction of 46 reductions

	Min	Max	Median	Mean
Reduction factor (%)	10	97	69	61
Reduction time (s)	13	1110	116	257
Reduction attempts	8	789	85	151

This doesn't quite explain the large discrepancy between *CVC4* and *Yices2*, as they both execute over the SMT_QF_NIA theory. The results do show a fairly large time spent in execution for *CVC4*, therefore the difference might be due to implementation.

Overall, generation does provide a rather substantial overhead for testing, and we are aware of certain possible optimizations to be done in *MF++*. Otherwise, the time spent in compilation and execution is largely SUT-dependent.

Regarding the effectiveness of test case reduction, Table II shows performance data for 46 reduction sessions—reductions associated with 18 of the 21 coverage-guided tests of §V-C, each repeated three times (for the other three tests it proved difficult to subsequently build the precise version of the library that was used when the coverage point was first investigated). These experiments were executed on a similar setup as above, except with a Intel Core i7-4770 3.40 GHz CPU.

We observe that the various reduction factors we investigate are rather volatile — between 8 and 789 reduction attempts per session, and between a 10% to 97% reduction in size. This indicates that the underlying tests and SUTs heavily affect the reduction process, as one would expect. However, an average of a 61% reduction rate overall, with an average time of 151s, indicates our implementation of the reducer is effective at removing large swathes of unnecessary code. We also separately compute the total size shrinkage in bytes across all 18 tests, and observe a total reduction of 88%.

VI. Related Work

Metamorphic testing and the oracle problem. Metamorphic testing [6] aims to circumvent the oracle problem. The oracle problem is at the core of software testing and has received lot of attention, which we broadly categorize into works that propose (pseudo-) oracles for various domains (e.g., [33], [34], [35], [36], [37], [38], [39], [40]) and methods for generating, learning and improving oracles (e.g., [41], [42], [43], [44], [45], [46], [47]). See [5] for a recent survey on the oracle problem.

Metamorphic testing has been applied to many domains, including compilers and code generators [48], [49], [50], [51], machine learning [52], autonomous driving [53], stochastic optimization [54], simulation and modeling [55], [56], web services and systems [57], [58], and sentiment analysis [59] (see [60] for a survey on metamorphic testing). Our work is distinct because it is an instance of metamorphic *fuzzing*: we have a set of metamorphic relations (equivalent implementations of derived operations) that we compose in a randomized fashion to yield rich test cases. In contrast, most existing metamorphic testing approaches involve using a fixed set of metamorphic relations to generate a set of follow-up tests from a fixed test suite. The most closely-related work is methods for

metamorphic compiler testing that involve random application of semantics-preserving transformations [48], [50], [61], [62].

Our approach of declaring derived operations in terms of one another is related to *composition of metamorphic relations* [63], but more general: rather than merely providing the output of one metamorphic relation to another, our approach allows a derived operation to be invoked from anywhere in the implementation of another derived operation.

MF++, but not its design and implementation, is discussed in a case study on metamorphic approaches to fuzzing [64].

Property-based testing. Property-based testing [65], [66], [67], [68], [69] involves writing a fixed unit test for which certain inputs are left unspecified. Random generation is then used to search for inputs that make the unit test fail. Our approach is similar in that it uses random input generation, but fundamentally different in that we do not perform input generation with respect to a fixed unit test, but rather we *generate* the unit test by randomly constructing a sequence of derived operations and then expanding this sequence in multiple equivalent ways. It would be possible to encode this as property-based testing by generating random-yet-equivalent sequences of operations as part of the process of random input generation. This would merely amount to re-implementing our approach inside a property-based testing framework.

Redundancy-based oracles. Our use of equivalent implementations is related to work on synthesising test oracles by exploiting software *redundancy* [70], [71]. This line of work observes that many systems exhibit natural redundancy—e.g., the `put` and `putAll` methods on a multimap data structure are implemented in a fundamentally different manner, but should behave equivalently when `putAll` is used with a singleton set. Exploiting such redundancy allows methods to be cross-checked against one another. A key difference is that rather than seeking existing redundancy within an implementation, our approach involves the library developer explicitly writing multiple redundant implementations of each derived operation, giving them the freedom to exercise the library in ways that they predict might be interesting from a bug-finding perspective.

Another testing approach that exploits redundancy is the ASTOOT method for testing object-oriented programs [72]. Like our approach this involves testing equivalent sequences of operations with respect to a user-supplied equivalence check, but it is not based on randomized testing.

Coverage-guided test case reduction. The design of the *MF++* reducer is based on hierarchical delta debugging [18], a specialized form of delta debugging [17]. Like the C-Reduce tool [73], our reducer applies transformations that are aware of C++ syntax, but unlike C-Reduce our reducer is heavily tailored to the syntax of *MF++*-generated tests (whereas C-Reduce is a generic reducer for C/C++ programs). Reducing with respect to coverage points, rather than bugs, is an example of *cause reduction* [19], [20]. Coverage-guided reduction has been combined with fuzzing to generate conformance tests for the Vulkan graphics API [23], but unlike our approach this work required test oracles to be added manually.

Test suite generation. Many techniques have been proposed for automated generation of high coverage test suites, e.g., based on genetic algorithms [3], [74], feedback-directed random generation [4], [75] and symbolic execution [76], [77], [78]. A limitation common to these techniques is that generated tests do not have straightforward oracles. For example, Randoop [4] generates test suites that characterise what the system under test does *today* so that changes to this behaviour can be automatically identified, while test generation using genetic algorithms has proposed using mutated versions of the system under test to serve as an oracle [74]. In contrast, our approach provides a metamorphic oracle by construction. The trade-off is the one-off manual effort associated with using our approach.

Testing SMT solvers. Four of our case studies are SMT solving libraries. Recent works have focused specifically on testing SMT solvers. The *semantic fusion* technique [79] has discovered many in *Z3* and *CVC4*, and the STORM technique has found soundness bugs in a variety of SMT solvers [80]. These techniques are metamorphic in nature: they involve manipulating existing formulas to yield new formulas whose satisfiability can be based on that of the original formula.

VII. Conclusions and Future Work

Our approach to metamorphic fuzzing enables automated generation of library tests with in-built oracles, based on a specification that the library developer provides as a one-off manual effort. We have shown that our implementation of this approach in the *MF++* is effective at bug-finding, revealing 21 previously-unknown bugs across four of our case study libraries. We also present a novel method for coverage-guided test case generation, leveraging the test case reducer of *MF++* to yield small tests that can be added to library regression test suites, and we have been successful in integrating 21 test cases into open source library regression test suites so far.

In future we plan to apply *MF++* to additional libraries in more diverse domains; e.g., the *cairo* [81] graphics library has a different approach to maintaining and modifying its state than what we have seen in our existing case studies. There is also scope for deepening the specifications of our existing libraries, e.g., to consider a richer set of SMT theories.

In this work we have opted to build a practical tool that really works for C++ libraries, at the expense of requiring the user to put manual effort into writing equivalent implementations of derived operations. We are sceptical as to how practical it would be to relieve this manual effort, due to the difficulty of program analysis for full-blown languages such as C++. However, we believe there is potential for static and dynamic analysis to aid in *suggesting* possible equivalences to the user, as well as providing additional diagnostic support to help the user debug the manual ingredients that they provide.

Acknowledgments

Our thanks to John Wickerson for feedback on an earlier draft of this work. This work was supported by the EPSRC HiPEDS Centre for Doctoral Training (EP/L016796/1) and the IRIS EPSRC Programme Grant (EP/R006865/1).

REFERENCES

[1] M. Zalewski, "American fuzzy lop (afl)," *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[2] K. Serebryany, "Continuous fuzzing with libFuzzer and AddressSanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 2016, pp. 157–157.

[3] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *FSE*, T. Gyimóthy and A. Zeller, Eds. ACM, 2011, pp. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[4] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *OOPSLA Companion*, R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds. ACM, 2007, pp. 815–816. [Online]. Available: https://doi.org/10.1145/1297846.1297902

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2014.2372785

[6] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," The Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01, 1998.

[7] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[8] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*. Springer, 2008, pp. 337–340.

[9] C. W. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *CAV*. Springer, 2011, pp. 171–177. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_14

[10] B. Dutertre and L. De Moura, "The Yices SMT solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[11] R. Brummayer, A. Biere, and F. Lonsing, "BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking," in *BPR workshop at CAV*, 2008.

[12] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Mathematical Software – ICMS 2010*, K. Fukuda, J. v. d. Hoeven, M. Joswig, and N. Takayama, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302.

[13] D. G. Wonnacott, "A retrospective of the omega project," *rap. tech. HC-CS-TR-2010-01, Haverford College Computer Science*, 2010.

[14] S. Verdoolaege, "Presburger formulas and polyhedral compilation," 2016. [Online]. Available: https://lirias.kuleuven.be/retrieve/361209

[15] A. Lascu, "MF++ GitHub repository," 2022. [Online]. Available: https://github.com/0152la/mfpp

[16] ——, "Replication package," 2021. [Online]. Available: https://hub.docker.com/r/mfpp/icst

[17] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002. [Online]. Available: https://doi.org/10.1109/32.988498

[18] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *ICSE*. ACM, 2006, pp. 142–151. [Online]. Available: https://doi.org/10.1145/1134285.1134307

[19] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *ICST*. IEEE Computer Society, 2014, pp. 243–252. [Online]. Available: https://doi.org/10.1109/ICST.2014.37

[20] ——, "Cause reduction: delta debugging, even without bugs," *Softw. Test. Verification Reliab.*, vol. 26, no. 1, pp. 40–68, 2016. [Online]. Available: https://doi.org/10.1002/stvr.1574

[21] The Clang Team, "Clang 12 documentation - libtooling," 2021. [Online]. Available: https://clang.llvm.org/docs/LibTooling.html

[22] GCC, the GNU Compiler Collection, "`gcov`–a test coverage program," 2021. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[23] A. F. Donaldson, H. Evrard, and P. Thomson, "Putting randomized compiler testing into production (experience report)," in *ECOOP*, ser. LIPIcs, vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 22:1–22:29. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2020.22

[24] A. Lascu, "Z3 issue: Incorrect solver result with explicit constant propagation," 2019. [Online]. Available: https://github.com/Z3Prover/z3/issues/2096

[25] boehmes (GitHub username), "Z3 issue: Incorrect proof related to div," 2019. [Online]. Available: https://github.com/Z3Prover/z3/issues/2082

[26] A. Lascu, "Z3 issue: Different satisfiability for commutative !=," 2019. [Online]. Available: https://github.com/Z3Prover/z3/issues/2604

[27] ——, "Z3 issue: Unknown result with ^ operation," 2020. [Online]. Available: https://github.com/Z3Prover/z3/issues/4775

[28] S. Verdoolaege, "Integer set coalescing," 2015. [Online]. Available: https://lirias.kuleuven.be/retrieve/293569

[29] A. Lascu, "isl issue: is_equal check failed on expected equivalent input sets," 2018. [Online]. Available: https://groups.google.com/d/msg/isl-development/BjxxUFI410c/rYJWV9neAgAJ

[30] ——, "Yices 2 issue: BVXOR3 yields inconsistent result," 2021. [Online]. Available: https://github.com/SRI-CSL/yices2/issues/353

[31] ——, "Z3 issue: Asan heap-use-after-free with recent coverage test," 2021. [Online]. Available: https://github.com/Z3Prover/z3/issues/5493

[32] ——, "Comment on Z3 pull request: Add Ubuntu CMake coverage CI step," 2021. [Online]. Available: https://github.com/Z3Prover/z3/pull/5442#issuecomment-889503465

[33] A. Avancini and M. Ceccato, "Grammar based oracle for security testing of web applications," in *AST*. IEEE Computer Society, 2012, pp. 15–21. [Online]. Available: https://doi.org/10.1109/IWAST.2012.6228984

[34] G. Jahangirova, A. Stocco, and P. Tonella, "Quality metrics and oracles for autonomous vehicles testing," in *14th IEEE Conference on Software Testing, Verification and Validation, ICST 2021, Porto de Galinhas, Brazil, April 12-16, 2021*. IEEE, 2021, pp. 194–204. [Online]. Available: https://doi.org/10.1109/ICST49551.2021.00030

[35] D. S. Katz, C. Hutchison, M. Zizyte, and C. L. Goues, "Detecting execution anomalies as an oracle for autonomy software robustness," in *2020 IEEE International Conference on Robotics and Automation, ICRA 2020, Paris, France, May 31 - August 31, 2020*. IEEE, 2020, pp. 9366–9373. [Online]. Available: https://doi.org/10.1109/ICRA40945.2020.9197060

[36] L. Mariani, M. Pezzè, and D. Zuddas, "Augusto: exploiting popular functionalities for the generation of semantic GUI tests with oracles," in *ICSE*. ACM, 2018, pp. 280–290. [Online]. Available: https://doi.org/10.1145/3180155.3180162

[37] T. A. Walsh, G. M. Kapfhammer, and P. McMinn, "Automated layout failure detection for responsive web pages without an explicit oracle," in *ISSTA*. ACM, 2017, pp. 192–202. [Online]. Available: https://doi.org/10.1145/3092703.3092712

[38] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand, "SOFIA: an automated security oracle for black-box testing of sql-injection vulnerabilities," in *ASE*. ACM, 2016, pp. 167–177. [Online]. Available: https://doi.org/10.1145/2970276.2970343

[39] C. Wang, F. Pastore, and L. C. Briand, "Oracles for testing software timeliness with uncertainty," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, pp. 1:1–1:30, 2019. [Online]. Available: https://doi.org/10.1145/3280987

[40] L. Ran, C. E. Dyreson, A. A. Andrews, R. C. Bryce, and C. J. Mallery, "Building test cases and oracles to automate the testing of web database applications," *Inf. Softw. Technol.*, vol. 51, no. 2, pp. 460–477, 2009. [Online]. Available: https://doi.org/10.1016/j.infsof.2008.05.016

[41] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An empirical validation of oracle improvement," *IEEE Trans. Software Eng.*, vol. 47, no. 8, pp. 1708–1728, 2021. [Online]. Available: https://doi.org/10.1109/TSE.2019.2934409

[42] C. Menghi, S. Nejati, K. Gaaloul, and L. C. Briand, "Generating automated and online test oracles for simulink models with continuous and uncertain behaviors," in *ESEC/FSE*. ACM, 2019, pp. 27–38. [Online]. Available: https://doi.org/10.1145/3338906.3338920

[43] K. Frounchi, L. C. Briand, L. Grady, Y. Labiche, and R. Subramanyan, "Automating image segmentation verification and validation by learning test oracles," *Inf. Softw. Technol.*, vol. 53, no. 12, pp. 1337–1348, 2011. [Online]. Available: https://doi.org/10.1016/j.infsof.2011.06.009

[44] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *ESEC/FSE*. ACM, 2020, pp. 1178–1189. [Online]. Available: https://doi.org/10.1145/3368089.3409758

[45] J. Chen, Y. Bai, D. Hao, L. Zhang, L. Zhang, B. Xie, and H. Mei, "Supporting oracle construction via static analysis," in *ASE*. ACM, 2016, pp. 178–189. [Online]. Available: https://doi.org/10.1145/2970276.2970366

[46] W. B. Langdon, S. Yoo, and M. Harman, "Inferring automatic test oracles," in *10th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*. IEEE, 2017, pp. 5–6. [Online]. Available: https://doi.org/10.1109/SBST.2017.1

[47] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *ICST*. IEEE Computer Society, 2013, pp. 342–351. [Online]. Available: https://doi.org/10.1109/ICST.2013.13

[48] A. F. Donaldson and A. Lascu, "Metamorphic testing for (graphics) compilers," in *Proceedings of the 1st international workshop on metamorphic testing*, 2016, pp. 44–47.

[49] Q. Tao, W. Wu, C. Zhao, and W. Shen, "An automatic testing approach for compiler based on metamorphic testing technique," in *2010 Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 270–279.

[50] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 93:1–93:29, 2017. [Online]. Available: https://doi.org/10.1145/3133917

[51] M. Boussaa, O. Barais, G. Sunyé, and B. Baudry, "Leveraging metamorphic testing to automatically detect inconsistencies in code generator families," *Softw. Test. Verification Reliab.*, vol. 30, no. 1, 2020. [Online]. Available: https://doi.org/10.1002/stvr.1721

[52] C. Murphy, G. E. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," 2008.

[53] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *ASE*. ACM, 2018, pp. 132–142. [Online]. Available: https://doi.org/10.1145/3238147.3238187

[54] S. Yoo, "Metamorphic testing of stochastic optimisation," in *ICST*. IEEE Computer Society, 2010, pp. 192–201. [Online]. Available: https://doi.org/10.1109/ICSTW.2010.26

[55] K. Sim, W. Pao, and C. Lin, "Metamorphic testing using geometric interrogation technique and its application," *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, vol. 1, no. 2, pp. 91–95, 2005.

[56] J. Ahlgren, M. E. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, E. Meijer, S. Sapora, and J. Spahr-Summers, "Testing web enabled simulation at scale using metamorphic testing," in *ICSE SEIP*. IEEE, 2021, pp. 140–149. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP52600.2021.00023

[57] W. K. Chan, S. C. Cheung, and K. R. Leung, "Towards a metamorphic testing methodology for service-oriented software applications," in *Fifth International Conference on Quality Software (QSIC'05)*. IEEE, 2005, pp. 470–476.

[58] P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "Metamorphic security testing for web systems," in *ICST*. IEEE, 2020, pp. 186–197. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00028

[59] M. H. Asyrofi, I. N. B. Yusuf, H. J. Kang, F. Thung, Z. Yang, and D. Lo, "BiasFinder: Metamorphic test generation to uncover bias for sentiment analysis systems," *CoRR*, vol. abs/2102.01859, 2021. [Online]. Available: https://arxiv.org/abs/2102.01859

[60] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[61] A. F. Donaldson, P. Thomson, V. Teliman, S. Milizia, A. P. Maselco, and A. Karpinski, "Test-case reduction and deduplication almost for free with transformation-based compiler testing," in *PLDI*. ACM, 2021, pp. 1017–1032. [Online]. Available: https://doi.org/10.1145/3453483.3454092

[62] M. Windsor, A. F. Donaldson, and J. Wickerson, "C4: the C compiler concurrency checker," in *ISSTA*. ACM, 2021, pp. 670–673. [Online]. Available: https://doi.org/10.1145/3460319.3469079

[63] H. Liu, X. Liu, and T. Chen, "A new method for constructing metamorphic relations," 08 2012, pp. 59–68.

[64] A. Lascu, M. Windsor, A. F. Donaldson, T. Grosser, and J. Wickerson, "Dreaming up metamorphic relations: Experiences from three fuzzer tools," in *MET*. IEEE, 2021, pp. 61–68. [Online]. Available: https://doi.org/10.1109/MET52542.2021.00017

[65] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 46, no. 4, pp. 53–64, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1988042.1988046

[66] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with quviq quickcheck," in *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. ACM, 2006, pp. 2–10.

[67] M. Papadakis and K. Sagonas, "A proper integration of types and function specifications with property-based testing," in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*. ACM, 2011, pp. 39–50.

[68] D. R. MacIver. (2007) What is hypothesis? [Online]. Available: https://hypothesis.works/articles/what-is-hypothesis/

[69] D. Maciver and A. F. Donaldson, "Test-case reduction via test-case generation: Insights from the hypothesis reducer (tool insights paper)," in *ECOOP*, ser. LIPIcs, vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 13:1–13:27. [Online]. Available: https://doi.org/10.4230/LIPIcs.ECOOP.2020.13

[70] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-checking oracles from intrinsic software redundancy," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 931–942. [Online]. Available: https://doi.org/10.1145/2568225.2568287

[71] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, N. Perino, M. Pezzè, and P. Tonella, "Intrinsic software redundancy for self-healing software systems, automated oracle generation," in *Software Engineering & Management 2015*, ser. LNI, U. Aßmann, B. Demuth, T. Spitta, G. Püschel, and R. Kaiser, Eds., vol. P-239. GI, 2015, pp. 129–130. [Online]. Available: https://dl.gi.de/20.500.12116/2528

[72] R. Doong and P. G. Frankl, "The ASTOOT approach to testing object-oriented programs," *ACM Trans. Softw. Eng. Methodol.*, vol. 3, no. 2, pp. 101–130, 1994. [Online]. Available: https://doi.org/10.1145/192218.192221

[73] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 335–346.

[74] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 278–292, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.93

[75] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, May 2007, pp. 75–84.

[76] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[77] N. Tillmann and J. de Halleux, "Pex-white box test generation for .net," in *TAP*, ser. Lecture Notes in Computer Science, vol. 4966. Springer, 2008, pp. 134–153. [Online]. Available: https://doi.org/10.1007/978-3-540-79124-9_10

[78] E. Albert, P. Arenas, M. Gómez-Zamalloa, and J. M. Rojas, "Test case generation by symbolic execution: Basic concepts, a clp-based instance, and actor-based concurrency," in *SFM*, ser. Lecture Notes in Computer Science, vol. 8483. Springer, 2014, pp. 263–309. [Online]. Available: https://doi.org/10.1007/978-3-319-07317-0_7

[79] D. Winterer, C. Zhang, and Z. Su, "Validating smt solvers via semantic fusion," in *PLDI*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 718–730. [Online]. Available: https://doi.org/10.1145/3385412.3385985

[80] M. N. Mansur, M. Christakis, V. Wüstholz, and F. Zhang, "Detecting critical bugs in SMT solvers using blackbox mutational fuzzing," in *ESEC/FSE*. ACM, 2020, pp. 701–712. [Online]. Available: https://doi.org/10.1145/3368089.3409763

[81] Cairo, "Cairo 2D graphics library," 2021. [Online]. Available: https://www.cairographics.org/