Dreaming up Metamorphic Relations: Experiences from Three Fuzzer Tools

Andrei Lascu
Imperial College London
London, United Kingdom, SW7 2AZ
Email: andrei.lascu10@imperial.ac.uk

Matt Windsor
Imperial College London
London, United Kingdom, SW7 2AZ
Email: m.windsor@imperial.ac.uk

Alastair F. Donaldson
Imperial College London
London, United Kingdom, SW7 2AZ
Email: alastair.donaldson@imperial.ac.uk

Tobias Grosser
University of Edinburgh
Edinburgh, United Kingdom, EH1 2LX
Email: tobias.grosser@ed.ac.uk

John Wickerson
Imperial College London
London, United Kingdom, SW7 2AZ
Email: j.wickerson@imperial.ac.uk

***Abstract*—Metamorphic testing requires the availability of a suitable set of metamorphic relations (MRs) for the application domain of interest. A software testing practitioner interested in using metamorphic testing is thus blocked unless they can devise a suitable set of MRs. In this paper we offer some practical advice on sources of inspiration for MRs, based on our experience building three fuzzing tools based on metamorphic testing: *MF++*, which supports automated testing of C++11 libraries, *C4*, which tests concurrency support in C11 compilers, and *spirv-fuzz*, which aims to find bugs in compilers for the SPIR-V programming language (mainly used in computer graphics). The MRs we have devised have taken inspiration from three main sources: (1) careful study of specification documents related to the libraries and programming languages that these tools target, (2) consultation of prior work and discussion with domain experts, and (3) manual inspection of the results of automated code coverage analysis on the systems under test. We describe these sources of inspiration in detail, giving a range of concrete examples for each. We hope that this experience report will help to inform developers of future metamorphic testing tools as to the steps they can take to discover MRs in their domains of interest.**

***Index Terms*—metamorphic relations, fuzzing, metamorphic testing, compiler testing, library testing, C11, SPIR-V**

I. Introduction

Metamorphic testing [1] involves using *metamorphic relations* (MRs) to generate new test cases from existing ones. Informally, an MR describes how certain changes to the input of a system under test (SUT) are expected to change the SUT's output. As a simple example, if a word count program tells us that a text file T contains 100 words, then we should expect the program to tell us that a text file T' contains *at least* 100 words whenever T' has T as a substring.

More formally, a metamorphic relation (MR) is characterised by a pair (R, S) of binary relations with the property that for any test cases x and y, if $(x, y) \in R$ holds then $(f(x), f(y)) \in S$ should also hold, where f represents the SUT. In our word count example, we would instantiate R to $\{(T, T') \mid T \text{ is a substring of } T'\}$, and S to "$\leq$". A fault in the SUT is revealed if a metamorphic relation is violated; e.g. a fault in the word count program would be revealed if *adding* text to a file caused the reported number of words in the file to *decrease*. Metamorphic testing has gained a lot of attention in recent years (see [2] for a recent survey) because it circumvents the *oracle problem* [3]: for a given test input x we can apply an MR to obtain a related test input x' (such that $(x, x' \in R)$), and then check whether the results $f(x)$ and $f(x')$ are appropriately related (i.e., whether $(f(x), f(x')) \in S$), without needing to know the expected value of either $f(x)$ or $f(x')$.

The main barrier to using metamorphic testing in practice is the need to devise suitable MRs. Writing meaningful and effective MRs requires intimate knowledge of the problem domain of the SUT, including in-built assumptions and preconditions on which the SUT depends.

Over the last few years we have developed three tools that employ metamorphic testing:

- *MF++* [4], for automated testing of C++11 libraries,
- *C4* [5], which tests compilation of C11 concurrency, and
- *spirv-fuzz* [6], which tests compilers for the SPIR-V language [7] (mainly used in computer graphics).

Our tools employ *metamorphic fuzzing*: they take an input to the SUT along with a set of MRs, and produce a stream of follow-up test cases by applying randomised combinations of said MRs to the input.

In the design of each of these tools we have had to formulate suitable metamorphic relations. We have found the following three strategies to be effective:

- **studying specifications**: looking carefully at API and programming language documentation to pinpoint expected properties of or relationships between API functions and language features,
- **consulting prior work and domain experts**: studying the MRs used in prior related work, and talking to experts in the application area of the SUT, then distilling the gained knowledge into new MRs, and

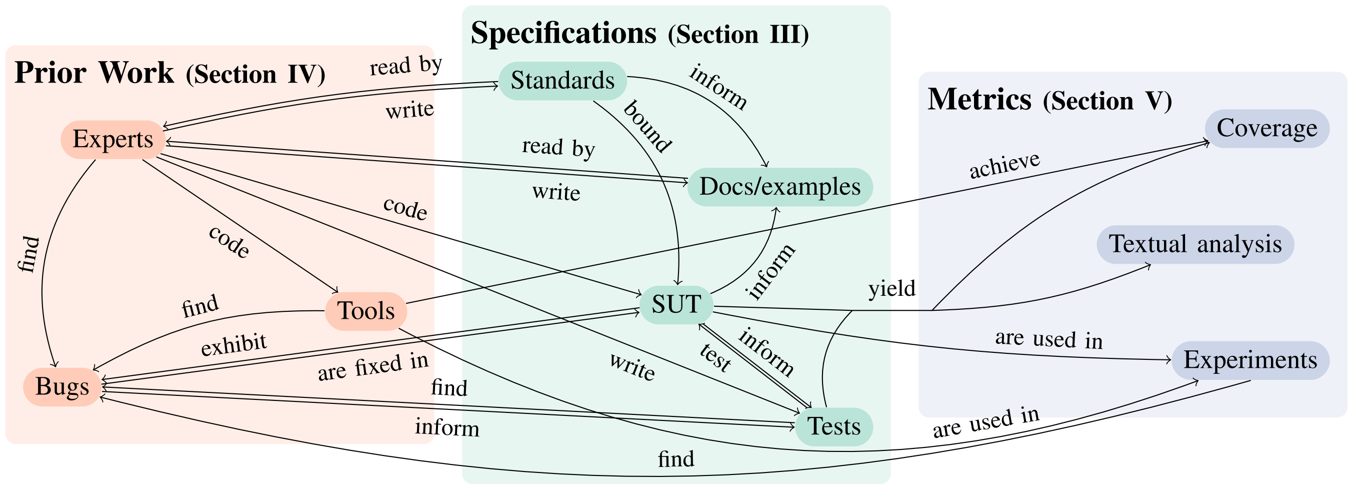

Fig. 1. Some of the inspiration sources and strategies we consider in this paper, and their relationships.

- **deriving metrics from the SUT**: collecting coverage information for the source code of the SUT based on running a metamorphic testing tool with a given set of MRs, or performing keyword-based searches of the SUT source code to identify key source files that appear relevant to the behaviours the fuzzing tool aims to exercise, then manually analysing the results and using blind spots as inspiration for new MRs that might alleviate them.

These strategies cover various levels of abstraction from the SUT, ranging from 'black-box' information about *expected* behaviour to 'white-box' inspection of *actual* behaviour; some sources operate on levels in-between. As we show in fig. 1, the strategies capture a network of deeply related sources of knowledge about the SUT. For example, domain experts both read and write standards and documentation, and often create the existing tools and bug reports that we study for MRs.

Our contribution: Although the MRs we have derived for *MF++*, *C4* and *spirv-fuzz* are domain-specific, we believe the strategies we used to derive them could be fruitfully employed by other researchers and practitioners interested in metamorphic testing. Our contribution is an example-driven experience report describing these strategies in detail, with illustrative examples. We hope this will serve as a useful cookbook of ideas for manually writing MRs.

Paper structure: We provide background on the fuzzing tools we have developed (section II) and then describe, with examples, our three main sources of inspiration for MRs: studying specifications (section III), consulting prior work and domain experts (section IV), and metrics (section V). We also discuss related work (section VI) and conclude with a discussion of future work (section VII).

II. Background

We provide a brief overview of *MF++* (section II-A), *C4* (section II-B) and *spirv-fuzz* (section II-C).

A. *MF++*

MF++ [4] provides a platform for C++ library developers to integrate metamorphic testing into their own workflow. MRs are used in *MF++* indirectly, meaning that end-users do not directly provide the pair (R, S) as described in section I. Rather, the user decides on a set of abstract *operations* that can be performed on the data types of their library. For each abstract operation they also provide a *family* of distinct but equivalent concrete implementations, each of which is a sequence of functions of the library under test. The tool randomly chooses a sequence σ of the abstract operations. It then expands σ into a set of n *concrete* sequences, $\{\sigma_1, \ldots, \sigma_n\}$, where each σ_i is derived from σ by repeatedly replacing an abstract operation with a randomly-selected concrete implementation. Every sequence σ_i is then executed on the library under test with respect to the same input. The results should all agree: mismatches either reveal faults in the library implementation (i.e., testing is successful), or that the concrete implementations the user has provided for an abstract operation are not actually equivalent (in which case they should be fixed and testing should be repeated).

For example, suppose we are interested in testing a library for manipulating a bitvector data type, `bit_vec`. A straightforward abstract operation that can be performed on a bitvector is the *identity* operation, which returns a bitvector unmodified. Suppose our library offers functions `rotate_left`, `rotate_right` and `size`, which rotate the bits of a bitvector and yield the number of bits in a bitvector. Two possible (non-trivial) abstract implementations of *identity* are:

```
bit_vec identity_rotate_fully(bit_vec bv) {
  return bv.rotate_left(bv.size());
}

bit_vec identity_rotate_back_forth(bit_vec bv) {
  size_t r = rand() % bv.size();
  return bv.rotate_left(r).rotate_right(r);
}
```

In terms of the formal definition of an MR, an input to the library under test is a pair (σ, x) where σ is a sequence of library calls and x is a data value that provides the necessary inputs to the calls in σ. The library implementation executes an input (σ, x) by invoking each call in σ in turn, drawing data from x. A family F of equivalent implementations of an abstract operation can be viewed as an MR (R, S) where:

$$\begin{aligned} R &= \{((\sigma_1, x), (\sigma_2, x)) \mid \sigma_1 \in F \wedge \sigma_2 \in F\} \\ S &= \qquad \text{equality} \end{aligned}$$

The MRs that *MF++* consumes are described in terms of calls to functions of the library's API. Understanding the API of a given library is, therefore, crucial to writing effective MRs. Because the MRs (i.e., families of equivalent implementations) used by *MF++* vary between libraries, they are provided by the user rather than being hard-coded in the tool. To evaluate *MF++* we have prepared sets of MRs targeting various libraries for solving mathematical constraints: integer set library (isl) [8], Z3 [9], Yices2 [10] and Boolector [11]. We draw on examples from these sets of MRs in this paper.

B. C4

C4 [5] mutates concurrent C test-cases. It aims to provide oracles for compiler testing (complex tests with known expected behaviour) by fuzzing them from smaller, exhaustively-simulated tests. For example, consider the following test:

```
              assume(*x == 0);
-----------------------------------------------
                          || r0 = atomic_load(x);
  atomic_store(x, 1);     || r1 = atomic_load(x);
-----------------------------------------------
              assert(r0 <= r1);
```

C4 might fuzz this test into the following:

```
            assume(*x == 0 && *t == 27);
-----------------------------------------------
if (*t <= 27)             || r0 = atomic_load(x);
  atomic_store(x, 1);     || r1 = atomic_load(x);
                          || if (*t > 27) r0 = 53;
-----------------------------------------------
              assert(r0 <= r1);
```

This is valid because the tests fix specific initial values for all variables, including the new variable at `*t`. While the new conditionals *theoretically* alter the test's behaviour, in practice `*t` is always 27, the conditional on the left always executes, and the conditional on the right never executes. As such, the two programs have equivalent behaviour over `*x`, `r0`, and `r1`.

For this method to work, a generated test must observationally refine any simulated outcomes of the test on which it is based, with respect to the set of variables that appear in the original test. As the tests are self-contained with fixed inputs, this set-up resembles *equivalence modulo inputs* testing [12].

C4 mutates tests by applying *actions*, randomly selected and instantiated from hard-coded templates. Actions transform one or more statements in the test to introduce control flow, variables, atomic actions, and so on. Each action a obeys the refinement condition above, inducing an MR (R, S) over compilers where the inputs (t, t') are the C tests and the outputs (c, c') are the corresponding compiled executables:

$$\begin{aligned} R &= \{\, (t, t') \mid t' = a(t)\} \\ S &= \{(c, c') \mid \forall o \in \text{obs}(c').\, \exists o' \in \text{obs}(c).\, o = o'|_{\text{var}(c)}\} \end{aligned}$$

Here, obs(c) is the set of final states reachable by executing c, and $o|_{\text{var}(c)}$ restricts o to those variables mentioned in c.

C4 does not test the system-under-test (the compiler) directly using the MRs. Instead, each test-case used as *C4* input contains an oracle in the form of a postcondition (for instance, a disjunction over all final states observed by exhaustively simulating the test-case with HERD [13]); compiler bugs may exist if the postcondition holds when simulating at the C level but not when running the compiled test-case. The MRs are soundness arguments, witnessing actions that *C4* can perform on the test-case while preserving the validity of its oracle. This said, the derivation of *C4* actions (and therefore MRs) usually proceeds as if *C4* were a traditional metamorphic testing tool.

C. spirv-fuzz

Standard Portable Intermediate Language-V (SPIR-V) [7] is a low-level intermediate language somewhat similar to LLVM intermediate representation [14], but designed specifically for programming heterogeneous many-core systems, with a particular focus on graphics processing units (GPUs). It is the language used for writing programmable shaders in the Vulkan programming model [15]. Every device that ships an implementation of Vulkan (which means virtually all desktops, laptops, smartphones and tablets, except those from Apple) comes with a graphics driver that includes a compiler from SPIR-V to the instruction set of the device's GPU.

The *spirv-fuzz* tool [6] follows in the footsteps of the *GraphicsFuzz* tool [16] by providing a means for testing SPIR-V compilers using a metamorphic approach. The tool is equipped with a large number of source-to-source *transformations*, each of which takes a SPIR-V program and produces an *equivalent* program that should yield the same outputs when applied to the same inputs. Given an original program, *spirv-fuzz* produces a *variant* program by applying many of these transformations in a randomised fashion, and then compares the results obtained by compiling and running both the original and variant programs. As with *C4*, this testing approach is inspired by the equivalence modulo inputs technique [12], and a *spirv-fuzz* transformation is somewhat analogous to a *C4* action. However, while a *C4* action should observationally refine a program, a *spirv-fuzz* transformation should lead to an observationally-equivalent program.

More formally, if we view the SUT—an implementation of SPIR-V—as taking a SPIR-V program P and an input x for P, and yielding the result of compiling P and running it on x, each transformation α employed by *spirv-fuzz* induces a metamorphic relation (R, S) where:

$$\begin{aligned} R &= \{((P, x), (P', x)) \mid P' = \alpha(P)\} \\ S &= \qquad \text{equality} \end{aligned}$$

III. Inspiration from specifications

Any *specification* to which the SUT is supposed to adhere is an obvious inspiration source. We define 'specification' loosely, including formal specifications as well as documentation, APIs, header files, examples, test suites, and the SUT source code itself. Here, we give examples of MRs derived from studying specifications associated with our tool domains.

Example 1: Adjusting non-functional hints. It is fairly common for an SUT to have certain parameters that only relate to non-functional properties of the system. The fact that adjusting such parameters should have no effect on the output of the SUT is a straightforward MR. Some SPIR-V instructions accept optional arguments that provide optimisation *hints* to the SPIR-V compiler, but that should have no functional impact on how the SPIR-V code behaves. Transformations that add or remove such hints are thus good candidates for MRs, to test that the hints indeed have no semantic impact.

For example, the `OpLoad` and `OpStore` memory instructions that load from and store to memory accept a set of "memory operand" flags. One such flag is called `NonTemporal`, about which the specification states [7, p. 95]: "Hints that the accessed address is not likely to be accessed again in the near future". As another example, a function declaration can be marked `Inline` to indicate [7, p. 92]: "Strong request, to the extent possible, to inline the function". The *spirv-fuzz* tool features a variety of simple transformations that toggle these hints and have been useful in triggering several bugs. □

Example 2: Removing assertion hints. SPIR-V also includes some hints that allow the programmer to communicate assertions to the compiler that it can use when optimising. A function can be marked `Pure`, indicating [7, p. 92]: "Compiler can assume this function has no side effect, but might read global memory or read through dereferenced function parameters", while a loop can be marked with `MinIterations` N, which is an "Unchecked assertion that the loop executes at least a given number of iterations" [7, p. 90]. The *spirv-fuzz* tool has transformations that *remove* such assertion hints if they are already present in a module, but should not *add* them unless it is able to prove that they actually hold. User-provided assertions are present in other programming languages and analogous concepts may apply in the context of other SUTs. Removal of such hints provides an easy MR, so long as their removal should not affect the functionality of the SUT. □

Example 3: Rewriting a simple operation as a special case of a more complex one. We have found several examples where a careful reading of SUT specification documents reveals that a simple operation can be viewed as a special case of a more complex or general operation. A natural MR is thus to rewrite instances of the simple operation accordingly, which may reveal faults if it turns out that one of the operations has not been implemented correctly. Such rewrites resemble the MRs derived from *redundancies* inherent in SUTs by Carzaniga et al. [17]. We describe an instance of this inspiration source in action for each of our three tools.

In accordance with specification documents related to C11 concurrency, *C4* can rewrite atomic load actions as read-modify-write (RMW) actions where the writing step is known, through algebra, to be idempotent. For example, $x + 0 = x$, so `atomic_fetch_add(x, y)` (atomically: read the value of `x` then add `y` to `x`) refines `atomic_load(x)` if `y` evaluates to 0. *C4* can generate reasonably complex expressions for `y`, which can themselves include idempotent RMWs. Deriving MRs in *C4* for such operations involved looking at a variety of specifications. First, the operation names suggest an analogy with operations with known right units (such as $+$ and 0). Second, derived language documentation (such as *cppreference* [18]) gives us sufficient detail about the semantics of each operation to confirm that using such units is sound. Third, scrutiny of the LLVM source code revealed a lowering pass for idempotent fetches that not only handles obvious cases like adding 0, but also cases like bitwise-ANDing with -1 that depend on two's-complement arithmetic. Given this, we expanded our support for generating such fetch operations to include `atomic_fetch_and(x, y)` where `y` evaluates to -1.

An example of this in *MF++* is rewriting the `mod` operation in *Z3* [9], which led to the discovery of a bug where *Z3* incorrectly determined a formula to be satisfiable.[1] By using the fact that division in *Z3*'s integer theory is implemented as integer division (as in the SMTLIB2 declaration of Integer Theory implementation [19]), we can rewrite $a \bmod b$ as $a - (b * (a/b))$ (with the appropriate non-zero checks).

Integer addition in SPIR-V follows two's complement semantics. A special instruction, `OpIAddCarry`, is provided to support the case where one wishes to know whether a given addition resulted in overflow [7, p. 186]. When applied to a pair of 32-bit integers a and b, the instruction returns a struct with two 32-bit integer fields, one containing the 32 low-order bits of the sum $a+b$, the other containing the carry bit: 1 if the addition overflowed and 0 otherwise. Transforming a regular addition instruction into an `OpIAddCarry` operation that discards the carry bit thus provides a straightforward MR. □

Example 4: Consulting derived documentation. By *derived documentation* we mean any material related to the SUT that indirectly relates to its official specification (if an official specification even exists), such as API references, additional wiki pages, and tutorials. These provide an approachable access point to a new tool or a new domain of interest. For *MF++* this is particularly important, as MRs are user-provided examples exercising the API of the library under test. As such, for a person initially unfamiliar with a library they might wish to test, having access to the API in a digestible format, as well as potentially annotated examples, is invaluable. Even for libraries with some degree of familiarity, referring to an API reference helps pinpoint potential MR candidate functions.

As discussed in Example 3, we found *cppreference* [18] to be a useful source of inspiration for *C4* MRs; in fact we found this derived document more useful than the C11 standard. □

[1] https://github.com/Z3Prover/z3/issues/2238

Discussion: We discussed the possibility of using the SUT source code as a specification in its own right. For mature SUTs with large codebases, however, it is practically unfeasible to refer to code alone, and SUT source is not always available. In section V we discuss the use of coverage information and basic textual analysis to help identify the parts of a large SUT that might be relevant as inspiration sources for new MRs. Additionally, when consulting the source code of one SUT to generate MRs, we must keep in mind that certain implementation choices might not generalise to the domain, and thus not hold in other SUTs from the same domain.

While we did not consider them here, SUT test suites could also be useful sources of inspiration — either in terms of kinds of MRs to create, or which parts of the SUT to stress. Test suites can also contain maintained and curated examples exposing important functionality of the SUT, often including tests introduced in response to bugs encountered by users.

IV. Inspiration from prior work

When considering applying metamorphic testing to an SUT, it is worth considering (a) whether metamorphic testing has been applied previously to a similar domain, in which case it might be possible to retarget the MRs used in prior work, and (b) whether MRs can be mined from previous experience gained from engineering and working with the SUT, which might be captured in a bug tracker or just in the minds of experts. We give examples of how MRs have been inspired by studying existing metamorphic testing tools, looking at existing SUT bug reports, and consulting domain experts.

Example 5: Using MRs from existing tools in new domains. Metamorphic relations often need to capture knowledge about the specific SUT domain. However, if metamorphic testing tools exist in nearby domains, we can profit from the MRs devised for those tools. While we are unlikely to find MRs that apply directly to our domain, we may be able to adapt or at least draw inspiration from these existing MRs.

An example from *C4* is a family of actions, transplanted from *GraphicsFuzz* [16], that insert 'dead' blocks. These are blocks that are statically known to be unreachable, such as the body of `if (false) { ... }`. Inside dead blocks, *C4* can generate code that would usually be ill-formed or fail refinement; this exposed a bug in the way pre-release versions of GCC handled loop overflow.[2] While the overall structure of this action family is general, details such as how to generate known-false expressions change from domain to domain.

MF++ is very amenable to adapting existing MRs to other libraries in the same domain, or even loosely related domains. As the MRs are written using the API of the libraries under test, it is fairly common for libraries within the same domain to have similar functions. For example, the *Yices2* [10] and *Boolector* [11] SMT solvers both support the theory of bitvectors. Their APIs share share a large number of bitvector operations, with identical or similar naming conventions. As such, once MRs have been defined for one of these libraries, search and replace would take care of the majority of changes require to target the other. □

```
int g1 = 1, g2 = 0;
void P0() {
   for (int l = 0; l != 4; l++) {
      if (g1) return;
      for (g2 = 0; g2 >= 26; ++g2);
   }
}
void P1() { g2 = 42; }
```

Fig. 2. Bug stimulus from Morisset et al. [20], which inspired *C4* loop actions.

Example 6: Deriving MRs from existing known bugs. Reports of existing bugs can help us to find new MRs. Often, we can extract minimal stimuli from said reports; these may reflect an MR or a class of inputs that a particular MR may be able to capture. In addition, we can validate our tools by bringing such bugs into their search space, either by using an affected SUT version or by manually injecting them.

An example of such efforts on *C4* involves a bug originally described by Morisset et al. [20]. This bug involves two loops (fig. 2) that, at first, seem to execute a fixed number of times; in fact, the first loop always terminates immediately, and the second loop is unreachable. As such, the only code modifying `g2` is that which sets it to 42, and we should never observe any other value for `g2`; the bug exhibits when the compiler accidentally causes parts of the second loop to execute and modify `g2`. This bug inspired two new *C4* actions: one inserts arbitrary multiple-execution **`for`**-loops such as those seen in the bug; a second inserts **`break`** and **`continue`** statements, possibly wrapped in known-true **`if`** blocks, at the end of loops, producing pockets of dead code. □

Example 7: Devising MRs in collaboration with domain experts. *MF++* initially started out as a project to apply metamorphic testing to *integer set library* (*isl*) [8], before we decided to expand it to generic C++ libraries. A lot of work alongside *isl* developers went into polishing MRs used for testing, and to understand what parts of the libraries we should be focusing on. Over a period of 3 months of intense testing and collaboration, we were able to slowly polish existing MRs to uncover new bugs and improve our testing efficacy.

SPIR-V features various instructions to sample from images at specified coordinates. The specification includes language such as [7, p. 147]: *"`Coordinate` must be a scalar or vector of floating-point type. ... It may be a vector larger than needed, but all unused components appear after all used components."* An NVIDIA engineer filed a suggestion on the *spirv-fuzz* issue tracker,[3] stating: *"We had a bug report that ended up being triggered by an image instruction having unused components in the coordinate. ... Adding unused components seems ideally suited for the fuzzer."* Based on this suggestion from a domain expert, we were able to add a transformation to the tool that would rewrite relevant image instructions to introduce unused

[2]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=97501

[3]https://github.com/KhronosGroup/SPIRV-Tools/issues/3375

components. This is also an instance of devising an MR based on an existing known bug (as in Example 6), except that the bug and suggested MR were communicated to us via a domain expert. In this instance we were lucky that the domain expert proactively suggested the new transformation. More generally it is likely worth scheduling time to interview domain experts in order to elicit ideas for MRs. □

Example 8: Refining MRs based on false alarms. When introducing some MRs in *MF++* targeting *Z3* we experimented with the `POW` operation for integer exponentiation. By adding MRs including calls to the respective API function, but not limiting the operands in any way, we observed cases where changing an operand in a manner that we believed should have no effect would change the solver's result from SAT to UNKNOWN. After reporting it as a potential issue to the *Z3* developers,[4] we were made aware of a parameter which controls the limit of the exponent, above which the solver gives up. This allowed us to refine our MRs exercising the `POW` operation. However, our experience indicates how false alarms can stem from the metamorphic testing process, when assumptions are made which are not consistent with the tested features. This exposes parts of the SUT that could be better documented.

Discussion: Building on prior work has clear advantages, as illustrated by our examples, but may risk *limiting* the scope of a new metamorphic testing technique. For example, trying to devise MRs that allow particular known bugs to be reproduced might lead to a tool that is over-fitted for those sorts of bugs, such that when they are all fixed the tool has limited value. Furthermore, when employing any sort of fuzzing in testing, it is important to be aware that an SUT rapidly becomes immune to a particular fuzzer if the SUT developers are diligent about fixing fuzzer-found bugs [21]. As such, a metamorphic fuzzing tool that is over-dependent on existing testing work might inadvertently lead to the tool generating test cases to which well-tested SUTs are already immune.

An issue with taking inspiration from tools in a related but distinct problem domain is that we might overfit our tools to the wrong domain. For instance, we derived several relations in *C4* from relations in *GraphicsFuzz*, such as dead-code introduction, that we thought to be agnostic to the underlying programming language; any bugs found using such relations were usually sequential control-flow bugs, not concurrency-related bugs. This suggests an unintentional bias towards the types of bug expected in *GraphicsFuzz*'s domain despite a conscious attempt to generalise the relations to *C4*'s domain.

V. Inspiration from metrics

As well as the qualitative strategies discussed above, we can use data-driven strategies to help inspire MRs. We give some examples of how we have used text-based searching of SUT source code, as well as automated collection of statement coverage info, as a starting point for devising new MRs.

[4]https://github.com/Z3Prover/z3/issues/4815

```
bool b = true;
int exp = KNOWN_VALUE_OF(obj);
int x = RANDOM_EXPR;
b = compare_exchange_strong(&obj, &exp, x);
assert(b);
```

Fig. 3. *C4* device for injecting known-true using compare-exchange.

```
bool b = true; int exp = 42;
b = compare_exchange_strong(obj, &exp, des);
/* rewrite */ return (exp == 42);
/* to      */ return b;
```

Fig. 4. Optimisation valid for strong compare-exchanges, but not weak ones.

Example 9: Leveraging textual analysis and code coverage analysis. *C4* targets compilers, which usually have very large codebases. Because *C4* is focused on testing aspects of a compiler related to concurrency, it is expected that testing with *C4* might leave large parts of the total compiler code base uncovered. Yet if we could identify concurrency-related parts of the code base that are not well covered we might be able to devise new MRs that would remedy this (and potentially find more bugs in the long-run).

To investigate this idea we used textual analysis to help identify key files in the LLVM compiler that relate to concurrency. Specifically, we searched both filenames and source code in LLVM for keywords such as 'atomic', 'cmpxchg', and 'memory order', building a list of files and code lines that appear to be concurrency-related. We then compiled LLVM with coverage enabled, and ran *C4*-based testing for a number of hours. Having both data (via textual analysis) on which components of the compiler appeared to be concurrency-related, and data (via coverage analysis) on where *C4* was achieving poor coverage of these components, we were able to gain insights on new MRs that could be added to the tool to exercise the concurrency-related facilities of LLVM.

For example, *C4* initially had an action inserting `strong` compare-exchanges (with statically-guaranteed success) as shown in fig. 3. Strong compare-exchanges succeed provided that the expected value is the same as the object value, updating the object with the desired value when this occurs; the return value accurately reflects success. C also supports `weak` compare-exchanges, which may spuriously fail; *C4* did not support these, and missed any code that handled them (including conditions on rewrites that depend on `strong` semantics). Figure 4 shows one such rewrite, corresponding to an LLVM IR optimisation in `InstCombineCompares` that was not previously covered (and which we found using a combination of textual analysis and coverage as described above). The presence of such rewrites in LLVM led us to make *C4* generate both weak and failing compare-exchanges. □

Example 10: Combining code coverage with expert advice. When dealing with mature codebases for a less familiar SUT, it can be hard to act on code coverage information. Specifically, without expert knowledge it can be hard to know which sections of uncovered code in the SUT to focus on, or how to achieve additional coverage on partly covered source files. In Example V we discussed basic textual analysis as a possible solution to this problem. An alternative is to discuss coverage results with domain experts. We faced the challenges of an unfamiliar SUT when trying to expand testing of *isl* using *MF++*. At the advice of the *isl* developers, we decided to focus on the *coalesce* [22] operation: they advised that it was an operation that they considered hard to implement correctly, and they were concerned that it might be undertested. We shared *MF++*-based coverage data for the *coalesce* source file with the *isl* developers. They were quick to highlight what kind of operations we should include in our MRs in order to achieve the additional desired coverage. After implementing their recommendations, we were able to achieve nearly 100% statement coverage of the coalesce implementation, and found a previously-unknown bug due to the enhanced testing.[5] Like Example 7, this demonstrates that it can be fruitful to work with domain experts when seeking inspiration for MRs. □

Discussion: Although coverage analysis on relevant source files can identify parts of an SUT that are under-tested by a metamorphic testing tool, it may be far from obvious how to translate this back to new MRs. With *C4*, the optimisations we found from coverage analysis manipulated compiler intermediate representations (such as LLVM IR) or streams of generated assembly instructions. As *C4* operates at a level of abstraction much closer to C source code, we had to make educated guesses as to which C constructs would lower to representations that would trigger the optimisations.

The use of SUT source code interacts well with the use of coverage: the latter can be a useful guide to the former. For instance, *C4* targets large compiler codebases, where it is hard to see which parts of the code will yield useful metamorphic relations; coverage experiments, alongside sampling-based approaches (such as visiting files with names containing 'atomic') helped highlight parts of compilers that contain concurrency optimisations and support code.

We have focused on statement and branch coverage in our use of coverage metrics so far. These metrics have clear limitations in assessing the adequacy of a testing technique, and other stronger metrics have been proposed, such as *path coverage* [23]. However, automated tool support for stronger notions of coverage is not readily available.

VI. Related work

A recent survey on metamorphic testing provides a thorough overview of the field [2]. In this paper we have focused on methods to help gain inspiration for new MRs, which then need to be manually implemented in the relevant metamorphic testing tool. We discuss related work on deriving MRs, and—since *C4* and *spirv-fuzz* are both compiler testing tools—on metamorphic compiler testing.

Other metamorphic relation generation techniques: Existing MRs can be used as a building block to generate more complex MRs. One such instance is the notion of *composing metamorphic relations* [24], where two compatible MRs can be composed together, to form a new MR, encompassing the strengths of both original MRs. Another approach is using abstract knowledge to translate MRs between domains, via *metamorphic relation patterns* [25]. Methods for MR generation based on search algorithms [26], [27], machine learning [28] and symbolic execution [29] have also been investigated. These works are complementary to our contributions: we present a cookbook of ideas one might try when seeking inspiration for MRs, rather than a targeted approach for actually generating concrete MRs in a particular setting.

Metamorphic compiler testing tools: A recent survey on metamorphic testing includes a section on metamorphic compiler testing techniques [30]—*C4* and *spirv-fuzz* are both examples of metamorphic compiler testing tools. The idea of metamorphic compiler testing was proposed in the context of generating sets of programs that are equivalent by construction [31]. The *equivalence modulo inputs* (EMI) technique involves starting with an original program and generating equivalent programs by using coverage analysis to identify statements that are unreachable with respect to a given input and randomly removing them [12]; this can be viewed as a form of metamorphic testing. The CLsmith tool for OpenCL compiler testing takes an EMI-like approach, but introduces code that is dead by construction, rather than identifying existing dead code [32]. The *GraphicsFuzz* tool for testing of OpenGL shader compilers also involves generating families of equivalent programs by transforming an original program, but uses semantics-preserving program transformations that are based on static analysis, rather than taking the coverage-directed approach of the EMI technique [33], [16], and has been used as a basis for generating new conformance tests for the Vulkan programming model [34].

VII. Conclusions and Future Work

We have outlined the experiences that we, the developers of three tools based on metamorphic testing principles, have had with regards to the derivation of MRs. The sources of inspiration broadly fall into three categories, related to specifications, prior work, and metrics. Within each category we have given specific examples from our experience that we hope will be of interest to the metamorphic testing community, and that may prove useful in guiding developers of future metamorphic tools towards similar sources of inspiration.

While our tools took influence from parts of all three strategies, the differences in our approaches, targets, and contexts led to differences between the tools in the emphasis and specifics of each. For example, the MRs used by *MF++* to test the *isl* library benefited significantly from the advice of the *isl* developers, and expert advice was also beneficial in the design of *spirv-fuzz* transformations, but we have not yet consulted the developers of concurrency-related compiler features in relation to *C4*. Similarly, coverage analysis was used to drive the development of MRs in *C4* and *MF++* but not yet in *spirv-fuzz*. A clear future direction for us as

[5]https://groups.google.com/g/isl-development/c/BjxxUFI410c

the developers of these tools is to consider the sources of inspiration that have not yet been leveraged for a particular tool and see whether it proves useful as a source of new MRs. Furthermore, our use of metrics to inspire MRs for *C4* and *MF++* has so far been limited: for *C4* we have only looked at code coverage of the LLVM compiler (and not, for example, GCC), while in *MF++* we have only used coverage to assess the thoroughness with which the *isl* library is tested.

While we have tried to categorise our sources of MR inspiration, this is not a rigid taxonomy and is only based on our experiences developing *MF++*, *C4* and *spirv-fuzz*. We look forward to engaging with others in the metamorphic testing community to learn from their experiences, possibly with a view to writing a broader and more rigorous experience report.

VIII. Data availability

All three tools are hosted publicly on Github and are available in their respective repositories: *MF++* [4], *C4* [5] and *spirv-fuzz* [6]. As this paper is an experience report, there is no associated data set.

Acknowledgements

This work was supported by IRIS EPSRC Programme Grant (EP/R006865/1), EPSRC standard grant (EP/R011605/1), the EPSRC HiPEDS Centre for Doctoral Training (EP/L016796/1), and a grant from the Research Institute for Verified Trustworthy Software Systems (VeTSS) funded by the EPSRC.

References

[1] 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.

[2] 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.

[3] 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.

[4] A. Lascu, "MF++," 2021. Available: https://github.com/0152la/SpecAST

[5] M. Windsor and J. Wickerson, "The C4 concurrent C fuzzer," 2021. Available: https://github.com/c4-project/c4f

[6] Khronos Group, "spirv-fuzz, part of SPIR-V Tools," 2021. Available: https://github.com/KhronosGroup/SPIRV-Tools/#fuzzer

[7] J. Kessenich, B. Ouriel, and R. Krisch, "SPIR-V specification, version 1.5, revision 4, unified," October 2020, https://www.khronos.org/registry/spir-v/#spec.

[8] 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.

[9] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[10] B. Dutertre, "Yices 2.2," in *Computer Aided Verification - 26th International Conference, CAV 2014. Proceedings*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 737–744.

[11] R. Brummayer, A. Biere, and F. Lonsing, "BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking," in *Proceedings of the 1st International Workshop on Bit-Precise Reasoning, BPR*, 2008.

[12] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14.: Association for Computing Machinery, 2014, p. 216–226.

[13] J. Alglave and L. Maranget, "The diy7 tool suite," 2020. Available: http://diy.inria.fr

[14] The LLVM Project, "LLVM language reference manual," 2021, https://llvm.org/docs/LangRef.html.

[15] The Khronos Vulkan Working Group, "Vulkan 1.2 - a specification," 2020, https://www.khronos.org/registry/vulkan/specs/1.2/pdf/vkspec.pdf.

[16] 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.

[17] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-checking oracles from intrinsic software redundancy," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 931–942.

[18] cppreference.com, "Atomic operations library," 2020, https://en.cppreference.com/w/c/atomic.

[19] C. Tinelli, "SMTLIB2 ints theory declaration," 2015. Available: http://smtlib.cs.uiowa.edu/theories-Ints.shtml

[20] R. Morisset, P. Pawan, and F. Zappa Nardelli, "Compiler testing via a theory of sound optimisations in the C11/C++11 memory model," in *ACM Conf. on Programming Language Design and Implementation (PLDI)*. ACM, 2013.

[21] J. Regehr and A. Groce, "The saturation effect in fuzzing,", June 2020. Available: https://blog.regehr.org/archives/1796

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

[23] N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated test data generation using an iterative relaxation method," in *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '98/FSE-6. New York, NY, USA: Association for Computing Machinery, 1998, p. 231–244.

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

[25] Z. Q. Zhou, L. Sun, T. Y. Chen, and D. Towey, "Metamorphic relations for enhancing system understanding and use," *IEEE Transactions on Software Engineering*, 2018.

[26] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 701–712.

[27] A. Goffi, A. Gorla, A. Mattavelli, M. Pezzè, and P. Tonella, "Search-based synthesis of equivalent method sequences," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 366–376.

[28] U. Kanewala and J. M. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 1–10.

[29] G. Dong, T. Guo, and P. Zhang, "Security assurance with program path analysis and metamorphic testing," in *2013 IEEE 4th International Conference on Software Engineering and Service Science*. IEEE, 2013, pp. 193–197.

[30] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020.

[31] 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.

[32] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 65–76, 2015.

[33] A. F. Donaldson and A. 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*. ACM, 2016, pp. 44–47.

[34] A. F. Donaldson, H. Evrard, and P. Thomson, "Putting randomized compiler testing into production (experience report)," in *34th European Conference on Object-Oriented Programming*, ECOOP 2020, ser. LIPIcs, R. Hirschfeld and T. Pape, Eds., vol. 166. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 22:1–22:29.