Formal Verification of Developer Tests: a Research Agenda Inspired by Mutation Testing

Serge Demeyer[1,2][0000−0002−4463−2945], Ali Parsai[1][0000−0001−8525−8198], Sten Vercammen[1], Brent van Bladel[1], and Mehrdad Abdi[1]

[1] Universiteit Antwerpen, Belgium
[2] Flanders Make vzw, Belgium

Abstract. With the current emphasis on DevOps, automated software tests become a necessary ingredient for continuously evolving, high-quality software systems. This implies that the test code takes a significant portion of the complete code base—test to code ratios ranging from 3:1 to 2:1 are quite common.
We argue that "testware" provides interesting opportunities for formal verification, especially because the system under test may serve as an oracle to focus the analysis. As an example we describe five common problems (mainly from the subfield of mutation testing) and how formal verification may contribute. We deduce a research agenda as an open invitation for fellow researchers to investigate the peculiarities of formally verifying testware.

Keywords: Testware · Formal Verification · Mutation Testing.

1 Introduction

DevOps is defined by Bass *et al.* as "*a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality*" [6]. The combination of these practices enables a continuous flow, where the development and operations of software systems are combined in one seamless (automated) process. This allows for frequent releases to rapidly respond to customer needs. Tesla, for example, uploads new software in its cars once every month [30]. Amazon pushes new updates to production on average every 11.6 seconds [22].

The key to the DevOps approach is a series of increasingly powerful automated tests that scrutinise the commits. As a consequence, test code takes a significant portion of the complete codebase. Several researchers reported that test code is sometimes larger than the production code under test [13,43,48]. More recently, during a large scale attempt to assess the quality of test code, Athanasiou *et al.* reported six systems where test code takes more than 50% of the complete codebase [5]. Moreover, Stackoverflow posts mention that test to code ratios between 3:1 and 2:1 are quite common [3].

Knowing about the popularity of automated tests and the sheer size of resulting test suites, software engineers need tools and techniques to identify lurking faults in the test code. The "testware", as it is called, should be treated as a regular software system involving requirements, architecture, design, implementation, quality assurance, and—last but not least—maintenance [15]. Indeed, we have witnessed first-hand that not all software projects uphold graceful co-evolution between production code and test code [48]. This effectively means that the software is vulnerable for extended periods of time whenever the production code evolves but the test code does not follow (immediately).

$\Longrightarrow$ *Just like all software, testware would benefit from formal verification.*

Test code (unit-test code in particular) is written in standard programming languages, thus amenable to formal verification. It is therefore possible to augment test code with annotations (invariants, pre-conditions) and verify that certain properties hold: loop termination, post-conditions,

... [17,21]. Moreover, most test code follows a quite consistent structure: the *setup-stimulate-verify-teardown* (S-S-V-T) cycle [45]. The purpose of statements within the test code is therefore rather easy to deduce, making it possible to focus the verification process on the relevant test slices.

$\Longrightarrow$ *Test code is quite amenable to formal verification.*

The Reach–Infect–Propagate–Reveal criterion (a.k.a. the RIPR model) provides a fine-grained framework to assess effective tests, or, conversely, weaknesses in a test suite [27]. It states that an effective test should first of all *Reach* the fault, then *Infect* the program state, after which it should *Propagate* as an observable difference, and eventually *Reveal* the presence of a fault (probably via an assert statement).

$\Longrightarrow$ *The system under test provides an oracle for effective tests.*

In this position paper we argue that "testware" provides interesting opportunities for formal verification, especially because the system under test may serve as an oracle to focus the analysis. As an example we describe five common problems (mainly from the subfield of mutation testing) and how formal verification may contribute. We deduce a research agenda as an open invitation for fellow researchers to investigate the peculiarities of formally verifying testware.

The remainder of this paper is organised as follows. Section 2, provides the necessary background information on formal verification and mutation testing. Section 3 goes over the five items in the research agenda explaining the problem and how formal verification of the test code could alleviate the problem. Section 4 list a few papers which investigated how formal verification could help in analysing test programs. We conclude with an open invitation to the community in Section 5.

2 Background

2.1 Formal Specification and Verification

Formal verification and formal specification are two complementary steps, used when adopting formal methods in software engineering [18]. During formal specification one rigorously specifies what a software system ought to do, and afterwards, during formal verification, one uses mathematical proofs to show that the system indeed does so. It should come as no surprise that the two steps go hand in hand, as illustrated by the discovery of a bug in the JDK linked list [20]. In this paper we restrict ourselves to a particular kind of formal verification—the ones based on a tool tightly integrated with a normal programming language—exemplified by KeY [17] and VeriFast [21]. These tools insert special program statements (pragmas, annotations) into the code to express properties by means of invariants, pre-conditions, and post-conditions. A series of mathematical theorem provers are then used to show that these properties indeed hold.

2.2 Mutation Testing

Mutation testing (also called mutation analysis—within this text the terms are used interchangeably) is the state-of-the-art technique for evaluating the fault-detection capability of a test suite [23]. The technique deliberately injects faults into the code and counts how many of them are caught by the test suite. Within academic circles, mutation testing is acknowledged as the most effective technique for a fully automated assessment of the strength of a test suite. The most recent systematic literature survey by Papadakis *et al.* revealed more than 400 scientific articles between 2007 and 2016 investigating mutation testing from various angles [35]. Despite this impressive body of academic work, the technique is seldom adopted in an industrial setting because it comes with a tremendous performance overhead: each mutant must be compiled and tested separately [37]. During one of our experiments with an industrial codebase, we witnessed 48 hours of mutation testing time on a test suite comprising 272 unit tests and 5,258 lines of test code for a system under test comprising 48,873 lines of production code [46].

Example. Throughout this paper, we will use the C++ code in Figure 1 as a running example. It scans a vector from back to front, looking for an element. Upon finding the element, it returns its index (base zero) and -1 if the element is not found.

```
1 int findLast(std::vector<int> x, int y) {
2     if (x.size() == 0) return -1;
3     for (int i = x.size() - 1; i >= 0; i--)
4         if (x[i] == y)
5             return i;
6     return -1;
7 }
```

Fig. 1. C++ code searching for an element in a vector starting at the end

Now consider the test suite in Figure 2. The first test (`emptyVector`, lines 1 — 3) checks for the exceptional case of an empty vector. The second test (`doubleOccurrence`, lines 5 — 7), verifies the happy-day scenario: we look for an element in the vector and it should be found on position 3. This is a very relevant test because it actually looks for an element which appears two times in the vector and it correctly asserts that it should return the position of the last occurrence. The third test (`noOccurrence`, lines 9 — 11), checks what happens when we look for an element that is not in the vector, in which case it should return -1. Executing the test suite shows that all 3 tests pass. When calculating the code coverage, we even obtain a 100% statement, line and branch coverage.

```
 1 TEST(FindLastTests, emptyVector) {
 2   EXPECT_EQ(-1, findLast({}, 3));
 3 }
 4
 5 TEST(FindLastTests, doubleOccurrence) {
 6   EXPECT_EQ(3, findLast({1, 2, 42, 42, 63}, 42));
 7 }
 8
 9 TEST(FindLastTests, noOccurrence) {
10   EXPECT_EQ(-1, findLast({1, 2, 42, 42, 63}, 99));
11 }

[==========] 3 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.
```

Fig. 2. Test suite for the `findLast` in Figure 1

Terminology. As with every field of active research, the terminology is extensive. Below we list the most important terms readers should be familiar with.

Mutation Operators. Mutation testing mutates the program under test by artificially injecting a fault based on one of the known mutation operators. A mutation operator is a source code transformation which introduces a change into the program under test. Typical examples are replacing

a conditional operator (>= into <) or an arithmetic operator (+ into -). The first set of mutation operators were reported in King *et al.* [24]. Afterwards, special purpose mutation operators have been proposed to exercise novel language constructs, such as Java null-type errors [36] or C++11/14 lambda expressions and move semantics [38].

Killed and Survived (Live) Mutants. After generating the defective version of the software, the mutant is passed onto the test suite. If a test fails, the mutant is marked as killed (Killed Mutant). If all tests pass, the mutant is marked as survived or live (Survived Mutant).

Consider the mutated example in Figure 3, where we apply a mutation operator named "Relational Operator Replacement" (ROR). On line 3, >= is replaced by < and the complete test suite is executed. One test fails so the mutant is considered killed; the test suite was strong enough to detect this mutant.

```
1 int findLast(std::vector<int> x, int y) {
2     if (x.size() == 0) return -1;
3     for (int i = x.size() - 1; i < 0; i--)
4         if (x[i] == y)
5             return i;
6     return -1;
7 }
```

```
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] FindLastTests.doubleOccurrence
```

Relational Operator Replacement (ROR): On line 3, `i >= 0` is replaced by `i < 0`. At least one test fails, so the mutant is killed.

Fig. 3. Killed mutant in `findLast` from Figure 1

We again apply a "Relational Operator Replacement" (ROR), this time replacing >= by > and arriving at the situation in Figure 4. If we execute the complete test suite, all tests pass so the test suite needs to be strengthened to detect this mutant.

```
1 int findLast(std::vector<int> x, int y) {
2     if (x.size() == 0) return -1;
3     for (int i = x.size() - 1; i > 0; i--)
4         if (x[i] == y)
5             return i;
6     return -1;
7 }
```

```
[==========] 3 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.
```

Relational Operator Replacement (ROR): On line 3, `i >= 0` is replaced by `i > 0`. No test fails, so the mutant is live.

Fig. 4. Survived mutant in `findLast` in Figure 1

Examining, why this mutant is not detected shows that the test suite fails to check for an important boundary condition: looking for an element which appears on the first position in the vector. If we add an extra test (see Figure 5) the test suite now detects the mutant (1 test fails, `occurrenceOnBoundary`). Now it is now capable of killing the mutant.

```
13 TEST(FindLastTests, occurrenceOnBoundary) {
14   EXPECT_EQ(0, findLast({1, 2, 42, 42, 63}, 1));
15 }

[==========] 4 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 3 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] FindLastTests.occurrenceOnBoundary
```

We add an extra test to the test suite in Figure 2 which tests for the boundary condition, an element to be found on the first position.

Fig. 5. Strengtened test suite for `findLast`

Mutation Coverage. The whole mutation analysis ultimately results in a score known as the mutation coverage: the number of mutants killed divided by the total number of non-equivalent mutants injected. A test suite is said to achieve full mutation test adequacy whenever it can kill all mutants, thus reaching a mutation coverage of 100%. Such test suite is called a mutation-adequate test suite.

Reach–Infect–Propagate–Reveal (RIPR). The Reach–Infect–Propagate–Reveal criterion (a.k.a. the RIPR model) provides a fine-grained framework to assess weaknesses in a test suite which are conveniently revealed by mutation testing [27]. It states that an effective test should first of all *Reach* the fault, then *Infect* the program state, after which it should *Propagate* as an observable difference, and eventually *Reveal* the presence of a fault (probably via an assert statement but this depends on the test infrastructure).

Consider the test suite in Figure 2 and Figure 5 together with the mutant that exposed the weakness in the test suite in Figure 4. The first test (`emptyVector`, lines 1 – 3) does not even reach the fault injected on line 2. The second test (`doubleOccurrence`, lines 5 – 7), *reaches* the fault because it executes the faulty `i > 0` condition, but does not infect the program state; so it cannot propagate nor reveal. The third test (`noOccurrence`, lines 9 – 11), infects the program state because it actually creates a state where the loop counter should have become 0, yet this is never propagated hence not revealed. It is only the fourth test (`occurrenceOnBoundary`, lines 13 – 15) which effectively infects the program state (`i` does not become 0), propagates to the output (returns -1) where it is revealed by the assert statement (expected 0).

Invalid Mutants. Mutation operators introduce syntactic changes, hence may cause compilation errors in the process. A typical example is the arithmetic mutation operator which changes a '+' into a '-'. This works for numbers but does not make sense when applied to the C++ string concatenation operator. If the compiler cannot compile the mutant for any reason, the mutant is considered invalid and is not incorporated into the mutation coverage.

Redundant ("Subsumed") Mutants. Sometimes there is an overlap in which tests kill which mutants, hence some mutants may be considered redundant. Redundant mutants are undesirable, since they waste resources and add no value to the process. In addition, they inflate the mutation score because often it is easy to kill many redundant mutants just by adding a single test case. To

express this redundancy more precisely, the mutation testing community defined the concept of *subsuming* mutants.

Take for instance the mutant in Figure 6, which replaces `x[i] == y` with `x[i] != y` on line 4. It is an easy to kill mutant as it is killed by three tests (doubleOccurrence, noOccurrence and occurrenceOnBoundary). The mutant in Figure 6 is therefore said to be *subsumed by* the mutant in Figure 3. Any test in our test suite which kills the latter mutant (difficult) one will also kill the former (easy) one.

```
1 int findLast(std::vector<int> x, int y) {
2     if (x.size() == 0) return -1;
3     for (int i = x.size() - 1; i >= 0; i--)
4         if (x[i] != y)
5             return i;
6     return -1;
7 }
```

```
[  PASSED  ] 1 test.
[  FAILED  ] 3 tests, listed below:
[  FAILED  ] FindLastTests.doubleOccurrence
[  FAILED  ] FindLastTests.noOccurrence
[  FAILED  ] FindLastTests.occurrenceOnBoundary
```

Relational Operator Replacement (ROR): On line 4, `x[i] == y` is replaced by `x[i] != y`. This is a redundant mutant which is subsumed by the mutant in Figure 3 and Figure 4. Any test in our test suite which kills the mutant on line 2 will also kill the one on line 4.

Fig. 6. Redundant mutant for `findLast`

Equivalent Mutants. Some mutants do not change the semantics of the program, i.e. the output of the mutant is the same as the original program for any possible input. Therefore, no test case can differentiate between a so-called "equivalent mutant" and the original program. Equivalent mutants are not incorporated into the mutation coverage. Unfortunately, the detection of equivalent mutants is undecidable due to the halting problem. Therefore, it is left to the software engineer to manually weed out equivalent mutants.

Consider again the running example, now in Figure 7. This time we apply the Relational Operator Replacement (ROR) on line 2, replacing the `== 0` with `<= 0`. Executing the test suite shows that all test pass so at first glance we have a live mutant. However, a deeper analysis shows that since the size of a vector is always positive, the value of `== 0` will always be the same as `<= 0`. So there is no input we can provide to the program under test to kill this mutant. Thus, this is an equivalent mutant.

```
1 int findLast(std::vector<int> x, int y) {
2     if (x.size() <= 0) return -1;
3     for (int i = x.size() - 1; i >= 0; i--)
4         if (x[i] == y)
5             return i;
6     return -1;
7 }

[==========] 4 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 4 tests.
```

Relational Operator Replacement (ROR): On line 2, `== 0` is replaced by `<= 0`. Not a single test fails, so the mutant is live. But since the size of a vector is always positive, the value of `x.size() == 0` will always be the same as `x.size() <= 0`, regardless of the vector x. This mutant is therefore equivalent.

Fig. 7. Equivalent mutant of `findLast` from Figure 1

3 Research Agenda

3.1 Equivalent Mutants

Equivalent mutants have been heavily studied in the literature as they may induce heavy overhead on test engineers aiming for 100% mutation coverage [31]. The most pragmatic approach so far has been to compare the generated (byte) code of the mutated program against the original [34]. Due to compiler optimizations the syntactic differences between the original and mutated program may disappear and then they are considered *trivially* equivalent. This allows to identify the easy cases, however, for the difficult ones, further analysis is required.

A paper by Offutt *et al.* illustrates how program analysis can help to identify equivalent mutants by demonstrating that they belong to an *infeasible path* [33]. The authors argue that a mutant is equivalent if the injected mutant lies on an *infeasible path*, thus (according to the RIPR model) the injected mutant can never propagate to the assert statements that reveals it.

Research Agenda. We would even go one step further and use program verification to prove that a mutant is equivalent to the original. And if not, the counter example should provide us with an extra test that illustrate where they may differ, hence would strengthen the test suite even further.

To formally verify that a mutant is indeed equivalent we create a copy of the mutated function. Lines 1 —3 and lines 9—10 in Figure 8 show two version of findLast that can thus be tested by the same test suite. Then we rely on code coverage (which is easy to obtain) or program slicing to identify the assert statement in the unit tests that are affected. Inserting a post-condition on the assert expressions would allow to show that the mutant can never be revealed, thus is equivalent. Line 15 in Figure 8 added such a post-condition to an adapted version of the emptyVector test. It compares the result of the original method under test (`findLast`) with the mutated one (`findLastEquivalent`). If the program verifier shows that this post-condition actually holds, then we have shown that this is indeed an equivalent mutant. If not, the program verifier should give us a counter example which corresponds to a different execution path enforced by the mutant. This then provides a concrete execution path to create an additional test that highlights the difference.

> $\Longrightarrow$ *Formal verification may be helpful in confirming that a live mutant can never change the output of the system under test, thus is an equivalent mutant.*

3.2 Infinite loops

Some mutants induce an infinite loop into the program under test. Therefore, most mutation tools abort the program under test when it runs an order of magnitude longer than expected and mark

```
 1 int findLast(std::vector<int> x, int y) {
 2      if (x.size() == 0) return -1;

...

 9 int findLastEquivalentCandidate(std::vector<int> x, int y) {
10      if (x.size() <= 0) return -1;

 ...

11 TEST(FindLastTests, emptyVector) {
12      int res1, res2;
13      EXPECT_EQ(-1, res1 = findLast({}, 3));
14      EXPECT_EQ(-1, res2 = findLastEquivalent({}, 3));
15              //@ ensures res1 == res2;
16 }
```

Fig. 8. Inserting post-conditions to prove equivalence of mutant of `findLast` in Figure 1

the corresponding mutant as "killed". Note that this assumption is not always correct, as in rare occasions the mutant can take much longer to be analysed due to other circumstances. In such cases, the mutant should be counted as "survived", but automatic detection of these scenarios is undecidable due to the halting problem.

Research Agenda. To formally verify that a mutant is indeed causing an infinite loop, we would first do the mutation analysis as normal, thus aborting the program when it runs an order of magnitude longer than expected. However, we do not yet mark the mutant as *"killed"* but instead put it in a special category *"further analysis required"*. Next we would insert a trivial post-condition right after the injected mutant and use program verification to show that the loop before never terminates.

$\Longrightarrow$ *Formal verification may confirm that a mutant created an infinite loop, thus should be marked as "killed".*

3.3 Flaky Tests

Mutation testing assumes tests to be completely deterministic: every test run should produce the exact same output. However, there is the phenomenon of flaky tests: tests whose outcome can non-deterministically differ even when run on the same code under test [29]. When a test suite contains flaky tests, the mutation analysis is unpredictable, as some mutants might be killed when in fact the tests are failing due to flakiness and not the injected fault itself.

Shi *et al.* reported the first technique to tackle flaky tests during a mutation analysis [41]. When running each mutant-test pair, they keep track of whether the mutant is covered or not. When a mutant is not covered by a test, they mark the status as "unknown" and perform further analysis. Essentially they rerun the test suite multiple times to see whether the test coverage indeed changes.

Research Agenda. To formally verify that a mutant is suffering from flaky tests, we would extend the process described by Shi *et al.* with an extra step [41]. Once a potentially flaky test is identified, we would insert a trivial post-condition at the end of the test case and use program verification to show that the post-condition is not necessarily satisfied. Ideally, the verification would also provide a counter example, highlighting the program statements that cause the flaky behaviour.

$\Longrightarrow$ *Formal verification may identify the root cause of a mutant suffering from flaky tests.*

3.4 Test Clones

When two fragments of code are either exactly the same or similar to each other, we call them a code clone. A code clone is also synonymous with a software clone or duplicated code, and these terms can be used interchangeably. Code clones can be differentiated based on their degree of similarity. First, code clones can be divided into syntactic clones and semantic clones. Syntactic clones are code clones that are syntactically similar, and are further divided in three types: Type I, Type II, and Type III clones. Type I clones are exactly the same, only allowing differences in comments, whitespaces, and indentation. Type II clones are a little less strict than Type I clones as they also allow differences in variable names and literal values. Finally, Type III clones are even less strict than Type II clones. They also allow for lines of code to be added or removed in the clone fragment. Note that it is not required for these types of clones to be functionally similar. Semantic clones on the other hand are code clones that are semantically similar without necessarily being syntactically similar. They are often called Type IV clones and are the most challenging clones to detect.

A lot of research has already been performed on software clones. In 2007, Koschke performed a survey of the literature on software clones [26]. This was followed in 2009 by him and his colleagues (Roy *et al.*) with an extensive comparison and evaluation of all code clone detection techniques and tools [40]. Svajlenko *et al.* manually curated a data set containing six million inter-project clones (Type I, II, III, and IV), including various strengths of Type III similarity (strong, moderate, weak) [42]. Over the years, a lot of research has been performed to further investigate the prevalence, characteristics, impact, and detection methods of software clones. However, most of this research focuses on production code; test code is rarely ever considered separately [26], [40] [39].

In 2018, Hasanain *et al.* performed an industrial case study to better understand code clones (i.e. duplicated code) in test code. They used NiCad to detect clones on a large test suite provided by Ericsson and discovered that 49% (in terms of LOC) of the entire test code are clones [19]. In a follow-up study our lab confirmed the prevalence of clones in test code [8]. We observed between 23% and 29% test code duplication in three well-tested open source systems, which is significantly more than the average amount of clones found in production code (between 10% and 15%). Worse, we discovered that most of the clone detection tools suffer from false negatives (NiCad [10] = 83%, CPD–PMD [1] = 84%, iClones [16] = 21%, TCORE [7] = 65%), which leaves ample room for improvement.

Research Agenda. Mutation analysis can give an indication on duplicated test logic. By carefully analysing subsumption relationships between mutants, we can infer which tests are likely to target the same program logic, thus being so-called *semantic clones*, also known as *Type IV* clones. We would consider them candidate clones, likely to be part of the aforementioned false negatives. By inserting invariants at relevant locations, formal verification may give indications on why certain test clones go undetected.

$\Longrightarrow$ *Formal verification might indicate why certain test clones appear as false negatives.*

3.5 Test Amplification

Test amplification is the act of automatically transforming a manually written unit-test to exercise boundary conditions [11]. In that sense, test amplification is a special kind of test generation: it relies on test cases previously written by developers which it tries to improve.

DSpot is an example of a test amplification tool for Java projects [12] which has been replicated for Pharo/Smalltalk within our lab under the name of SmallAmp [2]. These tools combine two techniques: (i) evolutionary test case generation or *Input Amplification* [44], and (ii) regression oracle generation or *Assert Amplification* [47]. They iteratively create extra test cases by changing the setup and the assertions, resulting in a new and larger set of test cases. The tools rely on genetic algorithms to select tests which increase the mutation coverage, discarding others. This

process is performed for a fixed number of steps which eventually results in a new test suite, with a better mutation coverage than the initial one, thus covering more corner cases. In that sense, test amplification is a brute force approach which relies on machine learning techniques to select an optimal solution.

Research Agenda. Formal verification may be able to complement brute force test amplification. In a recent proof-of-concept we demonstrated that it is possible to amplify test cases with extra asserts for the *easy-to-kill* mutants [28]. The idea is inspired by dynamic program analysis and the RIPR model. We build a complete program trace of both the normal test execution and the mutated one. We then associate *easy-to-kill* mutants with test cases that reach, infect, propagate, yet do not reveal the fault. These are cases of missing assert statements and the tool prototype is capable of suggesting an assert statement to be added, even providing concrete values for the assert expressions. The *difficult-to-kill* mutants however require an in-depth investigation to understand why the fault does not infect the program state or why it does not propagate to the output. That is where formal verification may help. By adding a post-condition right after the infected statement the formal verification tool should be able to tell us whether the program state gets infected and whether the fault gets propagated. Inspecting the counter-examples generated by the theorem prover, we should be able to come up with extra statements in the test which would stimulate the unit under test to infect and propagate the fault.

$\Longrightarrow$ *Formal verification may help in generating stimuli on the unit under test for the difficult-to-kill mutants.*

4 Related Work

The relationship between mutation testing and formal verification has been explored before. Aichernig *et al.* [4] argue that tests can be generated from formally verified requirements, using mutation testing to supervise where to generate additional test cases. To avoid difficult to maintain test suites (such as cloned test code discussed in Section 3.4), they introduce the concept of abstract test cases which are refined into concrete ones and regenerated when appropriate. In a similar vein, Brillout *et al.* [9] generate test cases from Simulink models achieving a high mutation coverage. Nevertheless, all these approaches take the perspective of the system under test specified using some kind of formal model of its behaviour and using mutation testing to create a strong test suite.

We argue that the opposite angle is equally relevant: that one should apply formal verification on the test code itself. This angle remains largely unexplored, except for the problem of equivalent mutants (see Section 3.1). There, several authors already confirmed that formal verification indeed may help to detect equivalent mutants. Kintis *et al.* [25] exploited patterns of data flow to identify mutants that are equivalent to the original program for a specific subset of paths. Devroey *et al.* [14] assert that for finite behavioural models, the equivalent mutant problem can be transformed to the language equivalence problem of non-deterministic finite automata. Marcozzi *et al.* [32] attempt to prove the validity of logical assertions in the code under test. The technique is implemented in a tool that relies on weakest-precondition calculus and SMT solving for proving the assertions.

5 Conclusion

In this position paper we argue that "testware" provides interesting opportunities for formal verification, especially because the system-under-test may serve as an oracle to focus the analysis and reduce the search space. We described five common problems: (1) Equivalent Mutants; (2) Infinite Loops; (3) Flaky tests; (4) Test Clones; (5) Test Amplification; and explained how formal verification of the test-code could partially alleviate them. This results in a research agenda which serves as on open invitation for fellow researchers to investigate the peculiarities of formally analysing testware.

References

1. Finding duplicated code with CPD (2020), [on line] https://pmd.github.io/latest/pmd_userdocs_cpd.html — last accessed In July 2020
2. Abdi, M., Rocha, H., Demeyer, S.: Test amplification in the pharo smalltalk ecosystem. In: Proceedings IWST 2019 (International Workshop on Smalltalk Technologies). ESUG (2019)
3. Agibalov, A.: What is a normal "functional lines of code" to "test lines of code" ratio? (2015), [on line] https://softwareengineering.stackexchange.com/questions/156883/ — last accessed In August 2020
4. Aichernig, B.K., Lorber, F., Tiran, S.: Formal test-driven development with verified test cases. In: Proceedings MODELSWARD 2014 (2nd International Conference on Model-Driven Engineering and Software Development). pp. 626 – 635 (2014)
5. Athanasiou, D., Nugroho, A., Visser, J., Zaidman, A.: Test code quality and its relation to issue handling performance. IEEE Transactions on Software Engineering **40**(11), 1100–1125 (2014). https://doi.org/10.1109/TSE.2014.2342227
6. Bass, L., Weber, I., Zhu, L.: DevOps: A Software Architect's Perspective. Addison-Wesley Longman Publishing Co., Inc. (2015)
7. van Bladel, B., Demeyer, S.: A novel approach for detecting Type-IV clones in test code. In: Proceedings IWSC 2019 (IEEE 13th International Workshop on Software Clones). pp. 102–118. IEEE (2019). https://doi.org/10.1109/IWSC.2019.8665855
8. van Bladel, B., Demeyer, S.: Clone detection in test code: an empirical evaluation. In: Proceedings SANER 2020 (International Conference on Software Analysis, Evolution and Reengineering (SANER)). pp. 492–500. IEEE (2020). https://doi.org/10.1109/SANER48275.2020.9054798
9. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: de Boer, F., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) Proceedings FMCO 2009 (Formal Methods for Components and Objects). pp. 208–227. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
10. Cordy, J.R., Roy, C.K.: The NiCad clone detector. In: 2011 IEEE 19th International Conference on Program Comprehension. pp. 219–220. IEEE (2011)
11. Danglot, B., Vera-Perez, O., Yu, Z., Zaidman, A., Monperrus, M., Baudry, B.: A snowballing literature study on test amplification. Journal of Systems and Software **157** (2019)
12. Danglot, B., Vera-Pérez, O.L., Baudry, B., Monperrus, M.: Automatic test improvement with dspot: a study with ten mature open-source projects. Empirical Software Engineering, Springer Verlag (2019)
13. Daniel, B., Jagannath, V., Dig, D., Marinov, D.: Reassert: Suggesting repairs for broken unit tests. In: Proceedings ASE 2009 (International Conference on Automated Software Engineering). pp. 433–444. IEEE CS (2009). https://doi.org/10.1109/ASE.2009.17
14. Devroey, X., Perrouin, G., Papadakis, M., Legay, A., Schobbens, P.Y., Heymans, P.: Model-based mutant equivalence detection using automata language equivalence and simulations. Journal of Systems and Software **141**, 1 – 15 (2018). https://doi.org/10.1016/j.jss.2018.03.010
15. Fewster, M., Graham, D.: Software Test Automation: Effective Use of Test Execution Tools. ACM Press Series, Addison-Wesley (1999)
16. Göde, N., Koschke, R.: Incremental clone detection. In: 2009 13th European Conference on Software Maintenance and Reengineering. pp. 219–228. IEEE (2009)
17. Hähnle, R.: Quo vadis formal verification? In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.) Deductive Software Verification – The KeY Book: From Theory to Practice, pp. 1–19. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6_1
18. Hall, A.: Seven myths of formal methods. IEEE Software **7**(5), 11–19 (1990). https://doi.org/10.1109/52.57887
19. Hasanain, W., Labiche, Y., Eldh, S.: An analysis of complex industrial test code using clone analysis. In: Proceedings QRS 2018 (IEEE International Conference on Software Quality, Reliability and Security). pp. 482–489. IEEE (2018). https://doi.org/10.1109/QRS.2018.00061
20. Hiep, H.D.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M., de Gouw, S.: Verifying openjdk's linkedlist using key. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 217–234. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_13
21. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for c and java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_4

22. Jenkins, J.: Velocity culture (2011), keynote Address at the Velocity 2011 Conf.
23. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (sep 2011). https://doi.org/10.1109/TSE.2010.62
24. King, K.N., Offutt, A.J.: A fortran language system for mutation-based software testing. Software: Practice and Experience **21**(7), 685–718 (jul 1991). https://doi.org/10.1002/spe.4380210704
25. Kintis, M., Malevris, N.: Medic: A static analysis framework for equivalent mutant identification. Information and Software Technology **68**, 1 – 17 (2015). https://doi.org/10.1016/j.infsof.2015.07.009
26. Koschke, R.: Survey of research on software clones. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2007)
27. Li, N., Offutt, J.: Test oracle strategies for model-based testing. IEEE Transactions on Software Engineering **43**(4), 372–395 (2016). https://doi.org/10.1109/TSE.2016.2597136
28. Lu, Z.X., Vercammen, S., Demeyer, S.: Semi-automatic test case expansion for mutation testing. In: Proceedings VST 2020 (IEEE Workshop on Validation, Analysis and Evolution of Software Tests). pp. 1–7 (2020). https://doi.org/10.1109/VST50071.2020.9051637
29. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Proceedings FSE 2014 (22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering). pp. 643 – 453. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635920
30. M., R.: Everything you need to know about tesla software updates (2014), [on line] https://www.teslarati.com/everything-need-to-know-tesla-software-updates/ — last accessed In May 2020
31. Madeyski, L., Orzeszyna, W., Torkar, R., Jozala, M.: Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. IEEE Transactions on Software Engineering **40**(1), 23–42 (2014). https://doi.org/10.1109/TSE.2013.44
32. Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: Proceedings ICSE 2018 (40th International Conference on Software Engineering). pp. 456 – 467. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180191
33. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Software testing, verification and reliability **7**(3), 165–192 (1997). https://doi.org/10.1002/(SICI)1099-1689(199709)7:3<165::AID-STVR143>3.0.CO;2-U
34. Papadakis, M., Jia, Y., Harman, M., Le Traon, Y.: Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. pp. 936–946. IEEE Press, Piscataway, NJ, USA (2015). https://doi.org/10.1109/ICSE.2015.103
35. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Mutation testing advances: An analysis and survey. Advances in Computers **112**, 275 — 378 (2019). https://doi.org/10.1016/bs.adcom.2018.03.015
36. Parsai, A., Demeyer, S.: Do null-type mutation operators help prevent null-type faults? In: Catania, B., Královič, R., Nawrocki, J., Pighizzini, G. (eds.) Proceedings SOFSEM 2019 (Theory and Practice of Computer Science). pp. 419–434. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-10801-4_33
37. Parsai, A., Demeyer, S.: Comparing mutation coverage against branch coverage in an industrial setting. International Journal on Software Tools for Technology Transfer (may 2020). https://doi.org/10.1007/s10009-020-00567-y, https://doi.org/10.1007/s10009-020-00567-y
38. Parsai, A., Demeyer, S., Busser, S.D.: C++11/14 mutation operators based on common fault patterns. In: Medina-Bulo, I., Merayo, M.G., Hierons, R. (eds.) Proceedings ICTSS 2018 (IFIP International Conference on Testing Software and Systems). pp. 102–118. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-99927-2_9
39. Roy, C.K., Cordy, J.R.: Benchmarks for software clone detection: A ten-year retrospective. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (JSS). pp. 26–37. IEEE (2018)
40. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of computer programming **74**(7), 470–495 (2009)
41. Shi, A., Bell, J., Marinov, D.: Mitigating the effects of flaky tests on mutation testing. In: Proceedings ISSTA 2019 (the 28th ACM SIGSOFT International Symposium on Software Testing and

Analysis). pp. 112 – 122. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293882.3330568

42. Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M.M.: Towards a big data curated benchmark of inter-project code clones. In: 2014 IEEE International Conference on Software Maintenance and Evolution. pp. 476–480 (2014)
43. Tillmann, N., Schulte, W.: Unit tests reloaded: Parameterized unit testing with symbolic execution. IEEE Software **23**(4) (2006). https://doi.org/10.1109/MS.2006.117
44. Tonella, P.: Evolutionary testing of classes. In: Proceedings ISSTA 2004 (ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 119 – 128. Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/1007512.1007528
45. Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M.: On the detection of test smells: A metrics-based approach for general fixture and eager test. IEEE Transactions on Software Engineering **33**(12), 800–817 (2007). https://doi.org/10.1109/TSE.2007.70745
46. Vercammen, S., Demeyer, S., Borg, M., Eldh, S.: Speeding up mutation testing via the cloud: Lessons learned for further optimisations. In: Proceedings ESEM 2018 (12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement). pp. 26:1–26:9. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3239235.3240506
47. Xie, T.: Augmenting automatically generated unit-test suites with regression oracle checking. In: Thomas, D. (ed.) Proceedings ECOOP 2006 (Object-Oriented Programming). pp. 380–403. Springer Verlag, Berlin, Heidelberg (2006). https://doi.org/10.1007/11785477_23
48. Zaidman, A., Rompaey, B.V., van Deursen, A., Demeyer, S.: Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. International Journal on Empirical Software Engineering **16**(3), 325–364 (2011). https://doi.org/10.1007/s10664-010-9143-7