TU Delft

Delft University of Technology

# A snowballing literature study on test amplification

Danglot, Benjamin; Vera-Perez, Oscar; Yu, Zhongxing; Zaidman, Andy; Monperrus, Martin; Baudry, Benoit

**DOI**
10.1016/j.jss.2019.110398

**Publication date**
2019

**Document Version**
Accepted author manuscript

**Published in**
Journal of Systems and Software

**Citation (APA)**
Danglot, B., Vera-Perez, O., Yu, Z., Zaidman, A., Monperrus, M., & Baudry, B. (2019). A snowballing literature study on test amplification. *Journal of Systems and Software*, *157*, 1-16. Article 110398. https://doi.org/10.1016/j.jss.2019.110398

**Important note**
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

**Copyright**
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

**Takedown policy**
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

*This work is downloaded from Delft University of Technology.*
*For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.*

© 2019 Manuscript version made available under CC-BY-NC-ND 4.0 license https://
creativecommons.org/licenses/by-nc-nd/4.0/

# A Snowballing Literature Study on Test Amplification

Benjamin Danglot[1], Oscar Vera-Perez[1], Zhongxing Yu[3],
Andy Zaidman[2], Martin Monperrus[3], Benoit Baudry[3]
[1] INRIA, [2] TU Delft, [3] KTH Royal Institute of Technology

**Abstract:** The adoption of agile approaches has put an increased emphasis on testing, resulting in extensive test suites. These suites include a large number of tests, in which developers embed knowledge about meaningful input data and expected properties as oracles. This article surveys works that exploit this knowledge to enhance manually written tests with respect to an engineering goal (e.g., improve coverage or refine fault localization). While these works rely on various techniques and address various goals, we believe they form an emerging and coherent field of research, which we coin "test amplification". We devised a first set of papers from DBLP, searching for all papers containing "test" and "amplification" in their title. We reviewed the 70 papers in this set and selected the 4 papers that fit the definition of test amplification. We use them as the seeds for our snowballing study, and systematically followed the citation graph. This study is the first that draws a comprehensive picture of the different engineering goals proposed in the literature for test amplification. We believe that this survey will help researchers and practitioners entering this new field to understand more quickly and more deeply the intuitions, concepts and techniques used for test amplification.

**Keywords:** test amplification; test augmentation; test optimization; test regeneration; automatic testing

## 1. Introduction

Software testing is the art of evaluating an attribute or capability of a program to determine that it meets its required results [29].

With the advent of agile development methodologies, which advocate testing early and often, a growing number of software projects develop and maintain a test suite [36]. Those test suites are often large and have been written thanks to a lot of human intelligence and domain knowledge [80, 79]. Developers spend a lot of time in writing the tests [8, 9, 10], so that those tests exercise interesting cases (including corner cases), and so that an oracle verifies as much as possible the program behavior [30].

The wide presence of valuable manually written tests has triggered a new thread of research that consists of leveraging the value of existing manually-written tests to achieve a specific engineering goal. This is what we coin "test amplification". We introduce the term *amplification* as an umbrella for the various activities that analyze and operate on existing test suites and that are referred to as augmentation, optimization, enrichment, or refactoring in the literature.

The goal of this paper is to help this original research thread thrive. This has motivated us to conduct a survey of research literature so that existing research efforts are characterized,

can be compared and new research opportunities can be identified. Furthermore, it is our conjecture that with good foundations and maturation, test amplification has the potential to bring software testing to the next level in terms of efficiency and efficacy among practitioners by introducing new automatic processes that improve the manually written tests.

This paper studies the literature on test amplification. The reviewing methodology is based on backward- and forward- snowballing on the citation graph [32]. To the best of our knowledge, this review is the first that draws a comprehensive picture of the different engineering techniques and goals proposed in the literature for test amplification.

We structure our reviewed papers in four main categories, each of them being presented in a dedicated section. Section 3 presents techniques that synthesize new tests from manually-written tests. Section 4 focuses on the works that synthesize new tests dedicated to a specific change in the application code (in particular a specific commit). Section 5 discusses the less-researched, yet powerful idea of modifying the execution of manually-written tests. Section 6 is about the modification of existing tests to improve a specific property.

To sum up, our contributions are:

- The first ever snowballing literature review on test amplification
- The classification of the related work into four main categories to help newcomers in the field (students, industry practitioners) understand this body of work.
- A discussion about the outstanding research challenges of test amplification.

## 2. Method

In this section, we present the methodology of our systematic literature review.

### *2.1. Definition*

In this review, we use the following definition of test amplification:

> **Definition**: Test amplification consists of exploiting the knowledge of a large number of test cases, in which developers embed meaningful input data and expected properties in the form of oracles, in order to enhance these manually written tests with respect to an engineering goal (e.g., improve coverage of changes or increase the accuracy of fault localization).

*Example:* A form of test amplification is the addition of test cases automatically generated from the existing manual test cases to increase the coverage of a test suite over the main source code.

*Relation to related work:* Test amplification is complementary, yet, significantly different from most works on test generation. The key difference is what is given as input to the system. Most test generation tools take as input: the program under test or a formal specification of the testing property. **In contrast, test amplification is defined as taking as primary input test cases written by developers**.

### *2.2. Methodology*

Literature studies typically rigorously follow a methodology to ensure both completeness and replication. We refer to Cooper's book for a general methodological discussion on literature studies [18]. Specifically for the field of software engineering, well-known methodologies are systematic literature reviews (SLR) [34], systematic mapping studies (SMS) [53] and snowballing studies [63]. For the specific area of *test amplification*, we found that there is no consensus on the terminology used in literature. This is an obstacle to using the SLR and SMS methodologies, which both heavily rely on searching [15]. As snowballing studies are less subject to suffering from the use of diverse terminologies, we perform our study per Wohlin's guidelines [63, 32].

First, we run the search engine of DBLP for all papers containing "test" and "amplification" in their title (using stemming, which means that "amplifying" is matched as well). This has resulted in 70 papers at the date of the search (March 27, 2018)[1]. We have reviewed them one by one to see whether they fit in our scope according to the definition of subsection 2.1. This has resulted in four articles [26, 82, 35, 33], which are the seed papers of this literature study. The reason behind this very low proportion (4/70) is that most articles in this DBLP search are in the hardware research community, and hence do not fall in the scope of our paper.

We now briefly describe these four seed papers. More details are given in the following sections.

- [26] Hamlet and Voas introduce study how different testing planning strategies can amplify testability properties of a software system.
- [82] Zhang and Elbaum explore a new technique to amplify a test suite for finding bugs in exception handling code. Amplification consists in triggering unexpected exceptions in sequences of API calls.
- [35] Leung et al propose to modify the test execution by using information gathered from a first test execution. The information is used to derive a formal model used to detect data races in later executions.
- [33] Joshi et al try to amplify the effectiveness of testing by executing both concretely and symbolically the tests.

From the seed papers, we have performed a backward snowballing search step [32], i.e., we have looked at all their references, going backward in the citation graph. Two of the authors have reviewed the papers, independently. Then, these 2 authors cross-checked the outcome of their literature review, and kept each paper for which they both agreed that it fits the definition of test amplification (cf. subsection 2.1). Then, we have performed a forward literature search step, using the Google scholar search engine and "cited by" filter, from the set of papers, in order to find the most recent contributions in this area. A backward snowballing search step and a forward snowballing search step constitute what we call an "iteration". With

[1] the data is available at `https://github.com/STAMP-project/docs-forum/blob/master/scientific-data/`

each iteration, we select a set of papers for our study that we obtain through the snowballing action. We iterate until this set of selected paper is empty, i.e., when no paper can be kept, we stop the snowballing process in both ways: backward and forward.

Once we had selected the papers for our study, we distinguished 4 key approaches to amplification, which we use to classify the literature : amplification by Adding New Tests as Variants of Existing Ones (section 3); Amplification by Modifying Test Execution (section 5); Amplification by Synthesizing New Tests with Respect to Changes (section 4); Amplification by Modifying Existing Test Code (section 6). The missing terminological consensus mentioned previously prevented the design of a classification according to Petersen's guidelines [53]. Incrementally, we have refined the four categories by analyzing the techniques and goals in each paper. Our methodology is as follows: we assign a work to a category if the key technique of the paper corresponds to it, per a consensus between the authors. If no category captures the gist of the paper, we have created a new category. If two categories are found to be closely related, we merge both categories to create a new one. The incremental refinement of these findings led to the definition of four categories to organize this literature study.

### *2.3. Novelty*

There are a number of notable surveys in software testing [22, 39, 2]. However none of them is dedicated to test amplification. For instance, we refer to Edvardsson's et al's [22] and McMinn et al's [39] articles for a survey on test generation. Yoo and Harman have structured the work on test minimization, selection and prioritization [74] . In the prolific literature on symbolic execution for testing, we refer the reader to the survey of Păsăreanu and Visser [51].

In general, test optimization, test selection, test prioritization, test minimization, test reduction is out of the scope of this paper.

Similarly, the work on test refactoring is related, but not in scope. In particular, the work from Van Deursen et al. [61, 44] and Mesaros [40] focuses on improving the structural and diagnosability qualities of software tests, and is a mainly manual activity. In contrast, test amplification is meant to be fully automated, as other technical amplification such as sound amplification. Its goal is also different, in that its aim is to test more effectively with regard to a a given target criterion.

Harrold et al. [28] discusses the problem of "retesting software", where there is a section related to amplification. However, it is only a light account on the topic which is now outdated. To our knowledge, this survey is the first survey ever dedicated to test amplification.

Yusifoğlu et al. [78] discuss the new trends in software test-code engineering, and discuss the implications for researchers and practitioners in this area. To do this, they use a systematic mapping to identify areas that require more attention. Their work covers a larger scope than our work, since they study all software test-code engineering research, methods and empirical study, while we focus specifically on test amplification, with more depth.

## 3. Amplification by Adding New Tests as Variants of Existing Ones

The most intuitive form of test amplification is to consider an existing test suite, then generate variants of the existing test cases and add those new variants into the original test suite. We denote this kind of test amplification as $\text{AMP}_{\text{add}}$.

```
1 class Computer {
2     public void compute(int integer) {
3         if (integer > 2) {
4             return integer + 2;
5         } else {
6             return integer + 1;
7         }
8     }
9 }
```

Listing 1: Example of a toy method

```
1 @Test
2 public void test_compute() {
3     Computer computer = new Computer();
4     int actualValue = computer.compute(1);
5     assertEquals(2, actualValue);
6 }
```

Listing 2: Example of toy test method

**Definition: A test amplification technique $\text{AMP}_{\text{add}}$ consists of creating new tests from existing ones to achieve a given engineering goal. The most commonly used engineering goal is to improve coverage according to a coverage criterion.**

The works listed in this section fall into this category and have been divided according to their main engineering goal.

### *3.1. Example*

In this section we present an example of $\text{AMP}_{\text{add}}$ to illustrate this category of work. Let us consider the single Java method, presented in Listing 1.

This method contains an if statement. The conditional expression tests the value passed through the parameter. If the value is greater than 2, then the method returns the value plus 2, otherwise it returns the value plus 1. Applying $\text{AMP}_{\text{add}}$ requires to have existing tests. Consider the test method in Listing 2. This test method ensures the behavior of the program when the parameter is lower than 2, i.e., when the else branch of the if statement is executed.

According to this test, one can say that this program is "poorly" tested, since only one of the two branches is covered. One potential goal of an $\text{AMP}_{\text{add}}$ technique is to increase this branch coverage.

Now, an $\text{AMP}_{\text{add}}$ technique may be able to generate the amplified test method shown in Listing 3. The test Listing 3 is easily derivable from the existing test Listing 2 because only one literal and the assertion differ. This new test method executes the *then* branch of the if statement (see Listing 1 line 2 and 3) that was not executed before. That is to say, applying $\text{AMP}_{\text{add}}$ improves the test suite, by increasing the branch coverage of the program.

```java
@Test
public void amplified_test_compute() {
    Computer computer = new Computer();
    int actualValue = computer.compute(3);
    assertEquals(5, actualValue);
}
```

Listing 3: Example of amplified toy test method

### *3.2. Coverage or Mutation Score Improvement*

Baudry *et al.* [6] [5] improve the mutation score of an existing test suite by generating variants of existing tests through the application of specific transformations of the test cases. They iteratively run these transformations, and propose an adaptation of genetic algorithms (GA), called a bacteriological algorithm (BA), to guide the search for test cases that kill more mutants. The results demonstrate the ability of search-based amplification to significantly increase the mutation score of a test suite. They evaluated their approach on 2 case studies that are .NET classes. The evaluation shows promising results, however the result have little external validity since only 2 classes are considered.

Tillmann and Schulte [60] describe a technique that can generalize existing unit tests into parameterized unit tests. The basic idea behind this technique is to refactor the unit test by replacing the concrete values that appear in the body of the test with parameters, which is achieved through symbolic execution. Their technique's evaluation has been conducted on 5 .NET classes.

The problem of generalizing unit tests into parameterized unit tests is also studied by Thummalapenta et al. [38]. Their empirical study shows that unit test generalization can be achieved with feasible effort, and can bring the benefits of additional code coverage. They evaluated their approach on 3 applications from 1 600 to 6 200 lines of code. The result shows an increase of the branch coverage and a slight increase of the bug detection capability of the test suite.

To improve the cost efficiency of the test generation process, Yoo and Harman [75] propose a technique for augmenting the input space coverage of the existing tests with new tests. The technique is based on four transformations on numerical values in test cases, i.e., shifting ($\lambda$x.x$+1$ and $\lambda$x.x$-1$ ) and data scaling (multiply or divide the value by 2). In addition, they employ a hill-climbing algorithm based on the number of fitness function evaluations, where a fitness is the computation of the euclidean distance between two input points in a numerical space. The empirical evaluation shows that the technique can achieve better coverage than some test generation methods which generate tests from scratch. The approach has been evaluated on the triangle problem. They also evaluated their approach on two specific methods from two large and complex libraries.

To maximize code coverage, Bloem et al. [12] propose an approach that alters existing tests to get new tests that enter new terrain, i.e., uncovered features of the program. The approach first analyzes the coverage of existing tests, and then selects all test cases that pass a yet uncovered branch in the target function. Finally, the approach investigates the path conditions of the selected test cases one by one to get a new test that covers a previously uncovered branch. To vary path conditions of existing tests, the approach uses symbolic execution and

model checking techniques. A case study has shown that the approach can achieve 100% branch coverage fully automatically. They first evaluate their prototype implementation on two open source examples and then present a case study on a real industrial program of a Java Card applet firewall. For the real program, they applied their tool on 211 test cases, and produce 37 test cases to increase the code coverage. The diversity of the benchmark allows to make a first generalization.

Rojas et al. [57] have investigated several seeding strategies for the test generation tool Evosuite. Traditionally, Evosuite generates unit test cases from scratch. In this context, seeding consists in feeding Evosuite with initial material from which the automatic generation process can start. The authors evaluate different sources for the seed: constants in the program, dynamic values, concrete types and existing test cases. In the latter case, seeding analogizes to amplification. The experiments with 28 projects from the Apache Commons repository show a 2% improvement of code coverage, on average, compared to a generation from scratch. The evaluation based on Apache artifacts is stronger than most related work, because Apache artifacts are known to be complex and well tested.

Patrick and Jia [52] propose *Kernel Density Adaptive Random Testing* (KD-ART) to improve the effectiveness of random testing. This technique takes advantage of run-time test execution information to generate new test inputs. It first applies *Adaptive Random Testing* (ART) to generate diverse values uniformly distributed over the input space. Then, they use *Kernel Density Estimation* for estimating the distribution of values found to be useful; in this case, that increases the mutation score of the test suite. KD-ART can intensify the existing values by generating inputs close to the ones observed to be more useful or diversify the current inputs by using the ART approach. The authors explore the trade-offs between diversification and intensification in a benchmark of eight C programs. They achieve an 8.5% higher mutation score than ART for programs that have simple numeric input parameters, but their approach does not show a significant increase for programs with composite inputs. The technique is able to detect mutants 15.4 times faster than ART in average.

Instead of operating at the granularity of complete test cases, Yoshida et al. [76] propose a novel technique for automated and fine-grained incremental generation of unit tests through minimal augmentation of an existing test suite. Their tool, *FSX*, treats each part of existing cases, including the test driver, test input data, and oracles, as "test intelligence", and attempts to create tests for uncovered test targets by copying and minimally modifying existing tests wherever possible. To achieve this, the technique uses iterative, incremental refinement of test-drivers and symbolic execution. They evaluated *FSX* using four benchmarks, from 5K to 40K lines of code. This evaluation is adequate and reveals that FSX' result can be generalized.

### *3.3. Fault Detection Capability Improvement*

Starting with the source code of test cases, Harder et al. [27] propose an approach that dynamically generates new test cases with good fault detection ability. A generated test case is kept only if it adds new information to the specification. They define "new information" as adding new data for mining invariants with Daikon, hence producing new or modified invariants. What is unique in the paper is the augmentation criterion: helping an invariant inference technique. They evaluated Daikon on a benchmark of 8 C programs. These programs

vary from 200 to 10K line of code. It is left to future work to evaluate the approach on a real and large software application.

Pezze et al. [54] observe that method calls are used as the atoms to construct test cases for both unit and integration testing, and that most of the code in integration test cases appears in the same or similar form in unit test cases. Based on this observation, they propose an approach which uses the information provided in unit test cases about object creation and initialization to build composite cases that focus on testing the interactions between objects. The evaluation results show that the approach can reveal new interaction faults even in well tested applications.

Writing web tests manually is time consuming, but it gives the developers the advantage of gaining domain knowledge. In contrast, most web test generation techniques are automated and systematic, but lack the domain knowledge required to be as effective. In light of this, Milani et al. [41] propose an approach which combines the advantages of the two. The approach first extracts knowledge such as event sequences and assertions from the human-written tests, and then combines the knowledge with the power of automated crawling. It has been shown that the approach can effectively improve the fault detection rate of the original test suite. They conducted an empirical evaluation on 4 open-source and large JavaScript systems. Compared to related research, we note that it is original to consider JavaScript systems.

### *3.4. Oracle Improvement*

Pacheco and Ernst implement a tool called Eclat [47], which aims to help the tester with the difficult task of creating effective new test inputs with constructed oracles. Eclat first uses the execution of some available correct runs to infer an operational model of the software's operation. By making use of the established operational model, Eclat then employs a classification-guided technique to generate new test inputs. Next, Eclat reduces the number of generated inputs by selecting only those that are most likely to reveal faults. Finally, Eclat adds an oracle for each remaining test input from the operational model automatically. They evaluated their approach on 6 small programs. They compared Eclat's result to the result of JCrasher, a state of the art tool that has the same goal than Eclat. In their experimentation, they report that Eclat perform better than JCrasher: Eclat reveals 1.1 faults on average against 0.02 for JCrasher.

Given that some test generation techniques just generate sequences of method calls but do not contain oracles for these method calls, Fraser and Zeller [25] propose an approach to generate parametrized unit tests containing symbolic pre- and post-conditions. Taking concrete inputs and results as inputs, the technique uses test generation and mutation to systematically generalize pre- and post-conditions. Evaluation results on five open source libraries show that the approach can successfully generalize a concrete test to a parameterized unit test, which is more general and expressive, needs fewer computation steps, and achieves a higher code coverage than the original concrete test. They used 5 open-source and large programs to evaluate the approach. According to their observation, this technique is more expensive than simply generating unit test cases.

### *3.5. Debugging Effectiveness Improvement*

Baudry *et al.* [7] propose the test-for-diagnosis criterion (TfD) to evaluate the fault localization power of a test suite, and identify an attribute called Dynamic Basic Block (DBB)

to characterize this criterion. A Dynamic Basic Block (DBB) contains the set of statements that are executed by the same test cases, which implies all statements in the same DBB are indistinguishable. Using an existing test suite as a starting point, they apply a search-based algorithm to optimize the test suite with new tests so that the test-for-diagnosis criterion can be satisfied. They evaluated their approach on two programs: a toy program and a server that simulates business meetings over the network. These two programs are less than 2K line of code long, which can be considered as small.

Rö$\beta$ler et al. [56] propose BugEx, which leverages test case generation to systematically isolate failure causes. The approach takes a single failing test as input and starts generating additional passing or failing tests that are similar to the failing test. Then, the approach runs these tests and captures the differences between these runs in terms of the observed facts that are likely related with the pass/fail outcome. Finally, these differences are statistically ranked and a ranked list of facts is produced. In addition, more test cases are further generated to confirm or refute the relevance of a fact. It has been shown that for six out of seven real-life bugs, the approach can accurately pinpoint important failure explaining facts. To evaluate BugEx, they use 7 real-life case studies from 68 to 62K lines of code. The small number of considered bugs, 7, calls for more research to improve external validity.

Yu et al. [77] aim at enhancing fault localization under the scenario where no appropriate test suite is available to localize the encountered fault. They propose a mutation-oriented test case augmentation technique that is capable of generating test suites with better fault localization capabilities. The technique uses some mutation operators to iteratively mutate some existing failing tests to derive new test cases potentially useful to localize the specific encountered fault. Similarly, to increase the chance of executing the specific path during crash reproduction, Xuan et al. [73] propose an approach based on test case mutation. The approach first selects relevant test cases based on the stack trace in the crash, followed by eliminating assertions in the selected test cases, and finally uses a set of predefined mutation operators to produce new test cases that can help to reproduce the crash. They evaluated MuCrash on 12 bugs for Apache Commons Collections, which is 26 KLoC of source code and 29 KLoC of test code length. The used program is quite large and open-source which increases the confidence. but using a single subject is a threat to generalization.

### *3.6. Summary*

*Main achievements:* The works discussed in this section show that adding new test cases based on existing ones can make the test generation process more targeted and cost-effective. On the one hand, the test generation process can be geared towards achieving a specific engineering goal better based on how existing tests perform with respect to the goal. For instance, new tests can be intentionally generated to cover those program elements that are not covered by existing tests. Indeed, it has been shown that tests generated in this way are effective in achieving multiple engineering goals, such as improving code coverage, fault detection ability, and debugging effectiveness. On the other hand, new test cases can be generated more cost-effectively by making use of the structure or components of the existing test cases.

*Main Challenges:* While existing tests provide a good starting point, there are some difficulties in how to make better use of the information they contain. First, the number of new tests synthesized from existing ones can sometimes be large and hence an effective

strategy should be used to select tests that help to achieve the specific engineering goal; the concerned works are: [6, 5, 76]. Second, the synthesized tests have been applied to a specific set of programs and the generalization of the related approaches could be limited. The concerned works are: [60, 38, 75, 12, 52, 27, 47, 7, 56, 73]. Third, some techniques have known performance issues and do not scale well: [41, 25].

## 4. Amplification by Synthesizing New Tests with Respect to Changes

Software applications are typically not tested at a single point in time; they are rather tested incrementally, along with the natural evolution of the code base: new tests are typically added together with a change or a commit [80, 79], to verify, for instance, that a bug has been fixed or that a new feature is correctly implemented. In the context of test amplification, it directly translates to the idea of synthesizing new tests as a reaction to a change. This can be seen as a specialized form $\text{AMP}_{\text{add}}$, which considers a specific change, in addition to the existing test suite, to guide the amplification. We call this form of test amplification $\text{AMP}_{\text{change}}$.

**Definition: Test amplification technique $\text{AMP}_{\text{change}}$ consists of adding new tests to the current test suite, by creating new tests that cover and/or observe the effects of a change in the application code.**

We first present a series of works by Xu et al., who develop and compare two alternatives of test suite augmentation, one based on genetic algorithms and the other on concolic execution. A second subsection presents the work of a group of authors that center the attention on finding testing conditions to exercise the portions of code that exhibit changes. A third subsection exposes works that explore the adaptation and evolution of test cases to cope with code changes. The last subsection shows other promising works in this area.

### *4.1. Example*

Listing 4 shows a toy class and two test cases designed to verify its code. At some point in development, the code of the method is modified as shown in Listing 5. The change consists of the addition of a new block in line 6.

The existing test cases do not execute the new code. There is no test input in the $[3, 5]$ interval. An $\text{AMP}_{\text{change}}$ technique would increment the test suite with a new test case, like the one shown in Listing 6, that covers the new code. The technique should be able to generate an input that meets the requirement to reach the new or changed code and the right oracle given the new conditions.

### *4.2. Search-based vs. Concolic Approaches*

In their work, Xu et al. [69] focus on the scenario where a program has evolved into a new version through code changes in development. They consider techniques as (i) the identification of coverage requirements for this new version, given an existing test suite; and (ii) the creation of new test cases that exercise these requirements. Their approach first identifies the parts of the evolved program that are not covered by the existing test suite. In

```
class Computer{
  public int computeValue(int input) {
      if(input < 3) {
          return input/2;
      }
      return 0;
  }
}

class ComputerTest {
  int threshold = 4;

  @Test
  public testSmallInput() {
    Computer comp = new Computer();
    assertTrue(comp.computeValue(2) < threshold);
  }

  @Test
  public testDefault() {
    Computer comp = new Computer();
    assertEquals(comp.computeValue(10), 0);
  }
}
```

Listing 4: Initial version of a class and two test cases

```
class Computer{
  public int computeValue(int input) {
      if(input < 3) {
        return input/2;
      }
      if (input <= 5) {
        return 2*input;
      }
      return 0;
  }
}
```

Listing 5: Modified version of the initial class

```
@Test
public testInput() {
  Computer comp = new Computer();
  assertTrue(comp.computeValue(4) > threshold);
}
```

Listing 6: A test case that covers the new portion of code.

the same process they gather path conditions for every test case. Then, they exploit these path conditions with a concolic testing method to find new test cases for uncovered branches, analyzing one branch at a time.

Symbolic execution is a program analysis technique to reason about the execution of every path and to build a symbolic expression for each variable. Concolic testing also carries a symbolic state of the program, but overcomes some limitations of a fully symbolic execution by also considering certain concrete values. Both techniques are known to be computationally expensive for large programs.

Xu et al. avoid a full concolic execution by only targeting paths related to uncovered branches. This improves the performance of the augmentation process. They applied their technique to 22 versions of a small arithmetic program from the SIR [1] repository and achieved branch coverage rates between 95% and 100%. They also show that a full concolic testing is not able to obtain such high coverage rates and needs a significantly higher number of constraint solver calls.

In subsequent work, Xu et al. [65] address the same problem with a genetic algorithm. Each time the algorithm runs, it targets a branch of the new program that is not yet covered. The fitness function measures how far a test case falls from the target branch during its execution. The authors investigate if all test cases should be used as population, or only a subset related to the target branch or, if newly generated cases should be combined with existing ones in the population. Several variants are compared according to their cost in terms of test executions and their effectiveness, that is, whether the generated test cases achieve the goal of exercising the uncovered branches. The experimentation targets 3 versions of *Nanoxml*, an XML parser implemented in Java with more than 7 KLoC and included in the SIR [1] repository. The authors conclude that considering all tests achieves the best coverage, but also requires more computational effort. They imply that the combination of new and existing test cases is an important factor to consider in practical applications.

Xu et al. then dedicate a paper to the comparison of concolic execution and genetic algorithms for test suite amplification [68]. The comparison is carried out over four small (between 138 and 516 LoC) C programs from the SIR [1] repository. They conclude that both techniques benefit from reusing existing test cases at a cost in efficiency. The authors also state that the concolic approach can generate test cases effectively in the absence of complex symbolic expressions. Nevertheless, the genetic algorithm is more effective in the general case, but could be more costly in test case generation. Also, the genetic approach is more flexible in terms of scenarios where it can be used, but the quality of the obtained results is heavily influenced by the definition of the fitness function, mutation test and crossover strategy.

The same authors propose a hybrid approach [67]. This new approach incrementally runs both the concolic and genetic methods. Each round applies first the concolic testing and the output is passed to the genetic algorithm as initial population. Their original intention was to get a more cost-effective approach. The evaluation is done over three of the C programs from their previous study. The authors conclude that this new proposal outperforms the other two in terms of branch coverage, but in the end is not more efficient. They also speculate about possible strategies for combining both individual approaches to overcome their respective weaknesses and exploit their best features. A revised and extended version of this work is given in [66].

*4.3. Finding Test Conditions in the Presence of Changes*

Another group of authors have worked under the premise that achieving only coverage may not be sufficient to adequately exercise changes in code. Sometimes these changes manifest themselves only when particular conditions are met by the input. The following papers address the problem of finding concrete input conditions that not only can execute the changed code, but also propagate the effects of this change to an observable point that could be the output of the involved test cases. However, their work does not create concrete new test cases. Their goal is to provide guidance, in the form of conditions that can be leveraged to create new tests with any generation method.

It is important to notice that they do not achieve test generation. Their goal is to provide guidance to generate new test cases independently of the selected generation method.

Apiwattanapong et al. [3] target the problem of finding test conditions that could propagate the effects of a change in a program to a certain execution point. Their method takes as input two versions of the same program. First, an alignment of the statements in both versions is performed. Then, starting from the originally changed statement and its counterpart in the new version, all statements whose execution is affected by the change are gathered up to a certain distance. The distance is computed over the control and data dependency graph. A partial symbolic execution is performed over the affected instructions to retrieve the states of both program versions, which are in turn used to compute testing requirements that can propagate the effects of the original change to the given distance. As said before, the method does not deal with test case creation, it only finds new testing conditions that could be used in a separate generation process and is not able to handle changes to several statements unless the changed statements are unrelated. The approach is evaluated on Java translations of two small C programs (102 Loc and 268 LoC) originally included in the Siemens program dataset [31]. The authors conclude that, although limited to one change at a time, the technique can be leveraged to generate new test cases during regular development.

Santelices et al. [58] continue and extend the previous work by addressing changes to multiple statements and considering the effects they could have on each other. In order to achieve this they do not compute state requirements for changes affected by others. This time, the evaluation is done in one of the study subjects form their previous study and two versions of *Nanoxml* from SIR.

In another paper [59] the same authors address the problems in terms of efficiency of applying symbolic execution. They state that limiting the analysis of affected statements up to a certain distance from changes reduces the computational cost, but scalability issues still exist. They also explain that their previous approach often produces test conditions which are unfeasible or difficult to satisfy within a reasonable resource budget. To overcome this, they perform a dynamic inspection of the program during test case execution over statically computed slices around changes. The technique is evaluated over five small Java programs, comprising *Nanoxml* with 3 KLoC and translations of C programs from SIR having between 283 LoC and 478 LoC. This approach also considers multiple program changes. Removing the need of symbolic execution leads to a less expensive method. The authors claim that propagation-based testing strategies are superior to coverage-based in the presence of evolving software.

### *4.4. Other Approaches*

Other authors have also explored test suite augmentation for evolving programs with propagation-based approaches. Qui et al. [55] propose a method to add new test cases to an existing test suite ensuring that the effects of changes in the new program version are observed in the test output. The technique consists of a two step symbolic execution. First, they explore the paths towards a change in the program guided by a notion of distance over the control dependency graph. This exploration produces an input able to reach the change. In a second moment they analyze the conditions under which this input may affect the output and make changes to the input accordingly. The technique is evaluated using 41 versions of the *tcas* program from the SIR repository (179 LoC) with only one change between versions. The approach was able to generate tests reaching the changes and affected the program output for 39 of the cases. Another evaluation was also included for two consecutive versions of the *libPNG* library (28 KLoC) with a total of 10 independent changes between them. The proposed technique was able to generate tests that reached the changes in all cases and the output was affected in nine of the changes. The authors conclude that the technique is effective in the generation of test inputs to reach a change in the code and expose the change in the program output.

Wang et al. [62] exploit existing test cases to generate new ones that execute the change in the program. These new test cases should produce a new program state, in terms of variable values, that can be propagated to the test output. An existing test case is analyzed to check if it can reach the change in an evolved program. The test is also checked to see if it produces a different program state at some point and if the test output is affected by the change. If some of these premises do not hold then the path condition of the test is used to generate a new path condition to achieve the three goals. Further path exploration is guided and narrowed using a notion of the probability for the path condition to reach the change. This probability is computed using the distance between statements over the control dependency graph. Practical results of test cases generation in three small Java programs (from 231 LoC to 375 LoC) are exhibited. The method is compared to *eXpress* and *JPF-SE* two state of the art tools and is shown to reduce the number of symbolic executions by 45.6% and 60.1% respectively. As drawback, the technique is not able to deal with changes on more than one statement.

Mirzaaghaei *et al.* [42, 43] introduce an approach that leverages information from existing test cases and automatically adapts test suites to code changes. Their technique can repair, or evolve test cases in front of signature changes (*i.e.* changing the declaration of method parameters or return values), the addition of new classes to the hierarchy, addition of new interface implementations, new method overloads and new method overrides. Their effective implementation *TestCareAssitance* (TCA) first diffs the original program with its modified version to detect changes and searches in the test code similar patterns that could be used to complete the missing information or change the existing code. They evaluate TCA for signature changes in 9 Java projects of the Apache foundation and repair in average 45% of modifications that lead to compilation errors. The authors further use five additional open source projects to evaluate their approach when adding new classes to the hierarchy. TCA is able to generate test cases for 60% of the newly added classes. This proposal could also fall in the category of test repairing techniques. Section 6 will explore alternatives in a similar direction that produce test changes instead of creating completely new test cases.

In a different direction, Böhme et al. [13] explain that changes in a program should not be treated in isolation. Their proposal focuses on potential interaction errors between software changes. They propose to build a graph containing the relationship between changed statements in two different versions of a program and potential interaction locations according to data and control dependency. This graph is used to guide a symbolic execution method and find path conditions for exercising changes and their potential interactions and use a Satisfiability Modulo Solver to generate a concrete test input. They provide practical results on six versions the *GNU Coreutils* toolset that introduce 11 known errors. They were able to find 5 unknown errors in addition to previously reported issues.

Marinescu and Cadar [37] present a system, called *Katch*, that aims at covering the code included in a patch. Instead of dealing with one change to one statement, as most of the previous works, this approach first determines the differences of a program and its previous version after a commit, in the form of a code patch. Lines included in the patch are filtered by removing those that contain non-executable code (i.e., comments, declarations). If several lines belong to the same basic program block, only one of them is kept as they will all be executed together. From the filtered set of lines, those not covered by the existing test suite are considered as targets. The approach then selects the closest input to each target from existing tests using the static minimum distance over the control flow graph. Edges on this graph that render the target unreachable are removed by inspecting the data flow and gathering preconditions to the execution of basic blocks. To generate new test inputs, they combine symbolic execution with heuristics that select branches by their distance to the target, regenerate a path by going back to the point where the condition became unfeasible or changing the definition of variables involved in the condition. The proposal is evaluated using the *GNU findutils*, *diffutils* and *binutils* which are distributed with most Unix-based distributions. They examine patches from a period of 3 years. In average, they automatically increase coverage from 35% to 52% with respect to the manually written test suite.

A posterior work of the same group [48] also targets patches of code, focusing on finding test inputs that execute different behavior between two program versions. They consider two versions of the same program, or the old version with the patch of changed code, and a test suite. The code should be annotated in places where changes occur in order to unify both versions of the program for the next steps. Then they select from the test suite those test cases that cover the changed code. If there is no such test case, it can be generated using *Katch*. The unified program is used in a two stage dynamic symbolic execution guided by the selected test cases: look for branch points where two semantically different conditions are evaluated in both program versions; bounded symbolic execution for each point previously detected. At those points all possible alternatives in which program versions execute the same or different branch blocks are considered and used to make the constraint solver generate new test inputs for divergent scenarios. The program versions are then normally executed with the generated inputs and the result is validated to check the presence of a bug or an intended difference. In their experiments this validation is mostly automatic but in general should be performed by developers. The evaluation of the proposed method is based on the *CoREBench* [14] data set that contains documented bugs and patches of the *GNU Coreutils* program suite. The authors discuss successful and unsuccessful results but in general the tool is able to produce test inputs that reveal changes in program behaviour.

*4.5. Summary*

*Main achievements:* $\text{AMP}_{\text{change}}$ techniques often rely on symbolic and concolic execution. Both have been successfully combined with other techniques in order to generate test cases that reach changed or evolved parts of a program [67, 66, 37]. Those hybrid approaches produce new test inputs that increase the coverage of the new program version. Data and control dependency has been used in several approaches to guide symbolic execution and reduce its computational cost [13, 37, 62]. The notion of distance from statements to observed changes has been also used for this matter [37, 3].

*Main challenges:* Despite the progress made in the area, a number of challenges remain open. The main challenge relates to the size of the changes considered for test amplification: many of the works in this area consider a single change in a single statement [3, 55, 62]. While this is relevant and important to establish the foundations for $\text{AMP}_{\text{change}}$, this cannot fit current development practices where a change, usually a commit, modifies the code at multiple places at once. A few papers have started investigating multi-statement changes for test suite amplification [58, 37, 48]. Now, $\text{AMP}_{\text{change}}$ techniques should fit into the revision process and be able to consider a commit as the unit of change.

Another challenge relates to scalability. The use of symbolic and concolic execution has proven to be effective in test input generation targeting program changes. Yet, these two techniques are computationally expensive [69, 67, 66, 3, 58, 48]. Future works shall consider more efficient ways for exploring input requirements that exercise program changes or new uncovered parts. Santelices and Harrold [59] propose to get rid of symbolic execution by observing the program behavior during test execution. However, they do not generate test cases.

Practical experimentation and evaluation remains confined to a very small number of programs, in most cases less than five study subjects, and even small programs in terms of effective lines of code. A large scale study on the subject is still missing.

## 5. Amplification by Modifying Test Execution

In order to explore new program states and behavior, it is possible to interfere with the execution at runtime so as to modify the execution of the program under test.

**Definition: Test amplification technique $\text{AMP}_{\text{exec}}$ consists of modifying the test execution process or the test harness in order to maximize the knowledge gained from the testing process.**

One of the drawbacks of automated tests is the hidden dependencies that may exist between different unit test cases. In fact, the order in which the test cases are executed may affect the state of the program under test. A good and strong test suite should have no implicit dependencies between test cases.

The majority of test frameworks are deterministic, i.e., between two runs the order of execution of test is the same [50, 49].

An $\text{AMP}_{\text{exec}}$ technique would randomize the order in which the tests are executed to reveal hidden dependencies between unit tests and potential bugs derived from this situation.

*5.1. Amplification by Modifying Test Execution*

Zhang and Elbaum [82, 83] describe a technique to validate exception handling in programs making use of APIs to access external resources such as databases, GPS or bluetooth. The method mocks the accessed resources and amplifies the test suite by triggering unexpected exceptions in sequences of API calls. Issues are detected during testing by observing abnormal terminations of the program or abnormal execution times. They evaluated their approach on 5 Android artifacts. Their sizes vary from 6k to 18k line of codes, with 39 to 117 unit tests in the test suite. The size of the benchmark seems quite reasonable. The approach is shown to be cost-effective and able to detect real-life problems in 5 Android applications.

Cornu et al. [19] work in the same line of exception handling evaluation. They propose a method to complement a test suite in order to check the behaviour of a program in the presence of unanticipated scenarios. The original code of the program is modified with the insertion of `throw` instructions inside `try` blocks. The test suite is considered as an executable specification of the program and therefore used as an oracle in order to compare the program execution before and after the modification. Under certain conditions, issues can be automatically repaired by catch-stretching. The authors used nine Java open-source projects to create a benchmark and evaluate their approach. This benchmark is big enough to conclude the generalization of the results. The selected artifacts are well-known, modern and large: Apache artifacts, joda-time and so on. Their empirical evaluation shows that the short-circuit testing approach of exception contracts increases the knowledge of software.

Leung et al [35] are interested in finding data races and non-determinism in GPU code written in the CUDA programming language. In their context, test amplification consists of generalizing the information learned from a single dynamic run. The main contribution is to formalize the relationship between the trace of the dynamic run and statically collected information flow. The authors leverage this formal model to define the conditions under which they can generalize the absence of race conditions for a set of input values, starting from a run of the program with a single input. They evaluated their approach using 28 benchmarks in the NVIDIA CUDA SDK Version 3.0. They removed trivial ones and some of them that they cannot handle. The set of benchmarks is big enough and contains a diversity of applications to be convinced that the approach can be generalized.

Fang et al. [23] develop a performance testing system named *Perfblower*, which is able to detect and diagnose memory issues by observing the execution of a set of test cases. The system includes a domain-specific language designed to describe memory usage symptoms. Based on the provided descriptions, the tool evaluates the presence of memory problems. The approach is evaluated on 13 Java real-life projects. The tool is able to find real memory issues and reduce the number of false positives reported by similar tools. They used the small workload of the DaCapo [11] benchmark. They argue that developers will not use large workloads and it is much more difficult to reveal performance bugs under small workloads. These two claims are legit, however the authors do not provide any evidence of the scalability of the approach.

Zhang et al. [81] devise a methodology to improve the capacity of the test suite to detect regression faults. Their approach is able to exercise uncovered branches without generating new test cases. They first look for identical code fragments between a program and its previous version. Then, new variants of both versions are generated by negating branch conditions that force the test suite to execute originally uncovered parts. The behaviour of version variants

are compared through test outputs. An observed difference in the output could reveal an undetected fault. An implementation of the approach is compared with *EvoSuite* [24] in 10 real-life Java projects. In the experiments known faults are seeded by mutating the original program code. The results show that *EvoSuite* obtains better branch coverage, while the proposed method is able to detect more faults. The implementation is available in the form of a tool named *Ison*.

### *5.2. Summary*

*Main achievements:* $\text{AMP}_{\text{exec}}$ proposals provide cost-effective approaches to observe and modify a program execution to detect possible faults. This is done by instrumenting the original program code to place observations at certain points or mocking resources to monitor API calls and explore unexpected scenarios. It adds no prohibitive overheads to regular test execution and provides means to gather useful runtime information. Techniques in this section were used to analyze real-life projects of different sizes and they are shown to match other tools that pursue the same goal and obtain better results in some cases.

*Main challenges:* As shown by the relatively small number of papers discussed in this section, techniques for test execution modification have not been widely explored. The main challenge is to get this concept known so as to enlarge the research community working on this topic. The concerned works are: [82, 83, 19, 35, 23, 81].

## 6. Amplification by Modifying Existing Test Code

In testing, it is up to the developer to design integration (large) or unit (small) tests. The main testing infrastructure such as JUnit in Java does not impose anything on the tests, such as the number of statements in a test, the cohesion of test assertions or the meaningfulness of test methods grouped in a test class. In literature, there is work on modifying existing tests with respect to a certain engineering goal.

**Definition: Test amplification technique $\text{AMP}_{\text{mod}}$ refers to modifying the body of existing test methods. The goal here is to make the scope of each test cases more precise or to improve the ability of test cases at assessing correctness (with better oracles). Differently from $\text{AMP}_{\text{add}}$, it is not about adding new test methods or new tests classes.**

### *6.1. Example*

We now use an example to give an illustration of work in this category. Let us consider a simple Java class named *Stack* in Listing 7. The example is a simplified Java implementation of a stack that stores unique elements. In the implementation, the array *elems* contains the elements of the stack, and the *push* and *pop* functions represent the two standard push and pop stack operations. The functions *isFull* and *isEmpty* check whether the stack is full and empty respectively.

Given the Java class, existing automatic test-generation tools can generate a test suite for it. For instance, Listing 8 exemplifies a possible test generated by automatic test-generation

```
public class Stack {
    private Comparable[] elems;
    public Stack() { ... }
    public void push(Comparable i) { ... }
    public void pop() { ... }
    public boolean isFull() { ... }
    public boolean isEmpty() { ... }
}
```

Listing 7: Example of a toy class

```
public class StackTest {
  @Test
  public void test1() {
    Stack s1 = new Stack();
    s1.push('a');
    s1.pop();
 }
}
```

Listing 8: Initial test suite for the toy class

```
public  class  StackTest {
  @Test
  public  void  testAug1 () {
    Stack s1 = new  Stack();
    assertTrue(s1.isEmpty());
    assertFalse(s1.isFull());
    s1.push('a');
    assertFalse(s1.isEmpty());
    assertFalse(s1.isFull());
    s1.pop();
  }
}
```

Listing 9: Augmented test suite for the toy class

tools. Note however there are no assertions generated in the test suite. To detect problems during test execution, it typically relies on observing whether uncaught exceptions are thrown or whether the execution violates some predefined contracts.

A test amplification technique $\text{AMP}_{\text{mod}}$ may be able to generate the amplified test suite as shown in Listing 9. Compared with the original test suite, the augmented test suite has comprehensive assertions. These assertions reflect the behavior of the current program version under test and can be used to detect regression faults introduced in future program versions.

### *6.2. Input Space Exploration*

Dallmeier et al. [20] automatically amplify test suites by adding and removing method calls in JUnit test cases. Their objective is to produce test cases that cover a wider set of executions than the original test suite in order to improve the quality of models reverse engineered from the code. They evaluate *TAUTOKO* on 7 Java classes and show that it is able to produce richer typestates (a typestate is a finite state automaton which encodes legal usages of a class under test).

Hamlet and Voas [26] introduce the notion of "reliability amplification" to establish a better statistical confidence that a given software is correct. Program reliability is measured as the mean time to failure of the system under test. The core contribution relates reliability to testability assessment, that is, a measure of the probability that a fault in the program will propagate to an observable state. The authors discuss how different systematic test planning strategies, e.g., partition-based test selection [46], can complement profile-based test cases, in order to obtain a better measurement of testability and therefore better bounds to estimate the reliability of the program being tested.

### *6.3. Oracle Improvement*

Xie [64] amplifies object-oriented unit tests with a the technique that consists of adding assertions on the state of the receiver object, the returned value by the tested method (if it is a non-void return value method) and the state of parameters (if they are not primitive values). Those values depend on the behavior of the given method, which in turn depends on the state of the receiver and of arguments at the beginning of the invocation. The approach, named *Orstra*, consists of instrumenting the code and running the test suite to collect state of objects. Then, assertions are generated, which call observer methods (methods with a non-void return type, *e.g.*, toString()). To evaluate *Orstra*, the author uses 11 Java classes from a variety of sources. Theses classes are different in the number of methods and lines of code, and the author also uses two different third-party test generation tools to generate the initial test suite to be augmented. The results show that *Orstra* can effectively improve the fault-detection capability of the original automatically generated test suite.

Carzaniga *et al.* [17] reason about generic oracles and propose a generic procedure to assert the behavior of a system under test. To do so, they exploit the redundancy of software. Redundancy of software happens when the system can perform the same action through different executions, either with different code or with the same code but with different input parameters or in different contexts. They devise the notion of "cross-checking oracles", which compare the outcome of the execution of an original method to the outcome of an equivalent method. Such oracle uses a generic equivalence check on the returned values and the state of the target object. If there is an inconsistency, the oracle reports it, otherwise, the checking

continue. These oracles are added to an existing test suite with aspect-oriented programming. For the evaluation, they use 18 classes from three non-trivial open-source Java libraries, including Guava, Joda-Time, and GraphStream. The subject classes are selected based on whether a set of equivalences have already been established or could be identified. For each subject class, two kinds of test suites have been used, including hand-written test suites and automatically generated test suites by Randoop. The experimental results show that the approach can slightly increase (+6% overall) the mutation score of a manual test suite.

Joshi et al. [33] try to amplify the effectiveness of testing by executing both concretely and symbolically the tests. Along this double execution, for every conditional statement executed by the concrete execution, the symbolic execution generates symbolic constraints over the input variables. At the execution of an assertion, the symbolic execution engine invokes a theorem prover to check that the assertion is verified, according to the constraints encountered. If the assertion is not guaranteed, a violation of the behavior is reported. To evaluate their approach, the authors use 5 small and medium sized programs from SIR, including gzip, bc, hoc, space, and printtokens. The results show that they are able to detect buffer overflows but it needs optimization because of the huge overhead that the instrumentation add.

Mouelhi *et al.* [45] enhance tests oracles for access control logic, also called Policy Decision Point (PDP). This is done in 3 steps: select test cases that execute PDPs, map each of the test cases to specific PDPs and oracle enhancement. They add to the existing oracle checks that the access is granted or denied with respect to the rule and checks that the PDP is correctly called. To do so, they force the Policy Enforcement Point, *i.e.*, the point where the policy decision is setting in the system functionality, to raise an exception when the access is denied and they compare the produced logs with expected log. To evaluate, they conduct case studies on three Java applications developed by students during group projects. For these three subjects, the number of classes ranges form 62 to 122, the number of methods ranges from 335 to 797, and the number of lines of code ranges from 3204 to 10703. The experimental results show that compared to manual testing, automated oracle generation saves a lot of time (from 32 hours to 5 minutes).

Daniel *et al.* [21] devise *ReAssert* to automatically repair test cases, i.e., to modify test cases that fail due to a change. *ReAssert* follows five steps: record the values of failing assertions, re-executes the test and catch the failure exception, *i.e.*, the exception thrown by the failing assertion. From the exception, it extracts the stack trace to find the code to repair. Then, it selects the repair strategy depending on the structure of the code and on the recorded value. Finally, ReAssert re-compiles the code changes and repeats all steps until no more assertions fail. The tool was evaluated on six real and well known open source Java projects, namely *PMD*, *JFreeChart*, *Lucene*, *Checkstyle*, *JDepend* and *XStream*. The authors created a collection of manually written and generated tests cases by targeting previous versions of these programs. *ReAssert* was able to produce fixes from 25% to 100% of failing tests for all study subjects. An usability study was also carried out with two teams of 18 researchers working on three research prototypes. The participants were asked to accomplish a number of tasks to write failing tests for new requirements and code changes and were also asked to manually fix the failures. *ReAssert* could repair 98% of failures created by the participants' code changes. In 90 % of cases the repairs suggested by the tool matched the patches created by the participants. The authors explain that the success rate of the tool depends more on the structure of the code of the test than the test failure itself.

### 6.4. Purification

Xuan et al. [70] propose a technique to split existing tests into smaller parts in order to "purify" test cases. Here, purification can be seen as a form of test refactoring. A pure test executes one, and only one, branch of an if/then/else statement. On the contrary, an impure test executes both branches *then* and *else* of the same if/then/else statement in code. The authors evaluate their technique on 5 widely used open-source projects from code organizations such as Apache. The experimental results show that the technique increases the purity of test cases by up to 66% for if statements and 11% for try statement. In addition, the result also shows that the technique improves the effectiveness of program repair of Nopol [71].

Xuan *et al.* [72] aim at improving the fault localization capabilities by *purifying* test cases. By purifying, they mean to modify existing failing test cases into single assertion test cases and remove all statements that are not related to the assertion. They evaluated the test purification on 6 open-source java project, over 1800 bugs generated by typical mutation tool PIT and compare their results with 6 mature fault localization techniques. They show that they improve the fault localization effectiveness on 18 to 43% of all the faults, as measured per improved wasted effort.

### 6.5. Summary

*Main achievements:* What is remarkable in $\text{AMP}_{\text{mod}}$ is the diversity of engineering goals considered. Input space exploration provides better state coverage [20] and reliability assessment [26], oracle improvement allows to increase the efficiency and effectiveness of tests [64, 17, 33, 45, 21], test purification of test cases facilitate program repair [70] and fault localization [72].

*Main challenges:* Although impressive results have been obtained, no experiments have been carried out to study the acceptability and maintainability of amplified tests [20, 64, 26, 17, 33, 45, 21, 70, 72]. In this context, acceptability means that human developers are ready to commit the amplified tests to the version control system (e.g., the Git repository). The maintainability challenge is whether the machine-generated tests can be later understood and modified by developers. To our understanding, these are the main challenges of test code modification.

## 7. Analysis

Table 1 shows a summary of the results of our snowballing review. Every row corresponds to one iteration in the process. Column # **F-ref** shows the number of papers added by following forward references. Column # **B-ref** shows the number of papers added by following backward references. The last column shows the total number of papers added for each iteration.

| Step | # F-ref | # B-ref | Total |
|---|---|---|---|
| Seed | 0 | 0 | 4 |
| It 1 | 4 | 1 | 5 |
| It 2 | 0 | 3 | 3 |
| It 3 | 16 | 2 | 18 |
| It 4 | 6 | 2 | 8 |
| It 5 | 6 | 3 | 9 |
| It 6 | 2 | 0 | 2 |
| Total | 34 | 11 | 49 |

Table 1: Details on the snowballing review. The first column shows the iteration, the second column contains the number of forward reference retained at a given iteration, the third column shows the number of backward reference retained at a given iteration, and the fourth column contains the total number of reference retained at a given iteration. The first row corresponds to the starting set of reference, i.e., the seed references.

We now provide a recapitulation of all the dimensions considered in our study. This section provides an overall view on these papers, so that the reader can have a quick summary of the main lines of research that we analyzed.

### *7.1. Aggregated View*

Table 2 shows all the articles considered in this snowballing survey per our inclusion criteria. The first column of the table shows the citation information, as given in the "References" section. The second column shows the term that the authors use to designate the form of amplification that they investigate. Columns 3 to 18 are divided in three groups. The first group corresponds to the section in which we have included the paper in our survey. The second group corresponds to the different engineering goals that we have identified. The third group captures the different techniques used for amplification in each work. The final columns in the table contain the target programming language, the year and venue in which the paper has been published, the last name of the first author and the iteration of the snowballing process in which the paper was included in the study.

Each row in the table corresponds to a specific contribution.The rows are sorted first by the section in which the papers are included in our study, then by year and then by the last name of the first author. In total, the table contains 49 rows.

One can see that "augmentation" (15 contributions), "generation" (9 contributions) and "amplification" (7 contributions) are the terms that appear most frequently to describe the approaches reported here. Other similar terms such as "enrichment", "adaptation" and "regeneration" are used less frequently. Most proposals (19 contributions) focus on adding new test cases to the existing test suite. Test amplification in the context of a change or the modification of existing test cases have received comparable attention (16 and 14 contributions respectively). Some techniques that modify existing test cases also target the addition of new test cases (3 contributions) and amplify the test suite with respect to a change (3 contributions). Amplification by runtime modification is the least explored area.

Most works aim at improving the code coverage of the test suite (25 contributions). After that, the main goals are the detection of new faults and the improvement of observability (13 and 12 contributions respectively). Fault localization, repair improvement and crash reproduction receive less attention (4, 4 and 1 contributions respectively).

Table 2: List of papers included in this snowballing survey. The columns correspond to the article categorization, the engineering goals, techniques employed, the programming language of the systems under test and the publication details. The last column shows the iteration the paper was included in our study.

| Reference | Term used | Add new tests | With respect to change/diff | Runtime modification | Modifies existing tests | Improve coverage | Reproduce crashes | Detect new faults | Localize faults | Improve repair | Improve observability | Test code analysis | Application code analysis | Execution modification | Concolic execution | Symbolic execution | Search based heuristics | Target language | Venue | Publication year | Last name of first author | Iteration |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| [27] | augmentation | • | | | | | | • | | | | • | | | | | | C | ICSE | 2003 | Harder | 2 |
| [4] | optimization | • | | | | • | | | | | | • | | | | | • | .NET | ASE | 2002 | Baudry | 2 |
| [6] | optimization | • | | | | • | | | | | | • | | | | | • | Eiffel, C# | STVR | 2005 | Baudry | 3 |
| [5] | optimization | • | | | | • | | | | | | • | | | | | • | C# | IEEE Software | 2005 | Baudry | 3 |
| [47] | generation | • | | | | | | • | | | | • | | | | | | Java | ECOOP | 2005 | Pacheco | 1 |
| [7] | optimization | • | | | | | | | • | | | • | | | | | | Java | ICSE | 2006 | Baudry | 4 |
| [60] | generation | • | | | • | • | | | | | | • | | | | • | | Spec# | IEEE Software | 2006 | Tillmann | 5 |
| [38] | generalization | • | | | • | • | | | | | | • | | | | • | | C# | FASE | 2011 | Thummalapenta | 5 |
| [25] | generation | • | | | | | | • | | | | • | • | | | | | Java | ISSTA | 2011 | Fraser | 6 |
| [56] | generation | • | | | | | | | • | | | • | | | | | | Java | ISSTA | 2012 | Ropler | 5 |
| [75] | regeneration | • | | | | • | | | | | | • | | | | | • | Java | STVR | 2012 | Yoo | 4 |
| [54] | generation | • | | | | | | • | | | | • | | | | | | Java | ICST | 2013 | Pezze | 5 |
| [77] | augmentation | • | | | | | | | • | | | • | | | | | | Java | IST | 2013 | Yu | 5 |
| [12] | augmentation | • | | | | • | | | | | | • | | | | • | | C | QSIC | 2014 | Bloem | 3 |
| [41] | generation | • | | | | | | • | | | | • | | | | | | JavaScript | ASE | 2014 | Fard | 3 |
| [73] | mutation | • | | | | | • | | | | | • | | | | | | Java | ESEC/FSE | 2015 | Xuan | 3 |
| [57] | generation | • | | | | • | | | | | | • | | | | | • | Java | STVR | 2016 | Rojas | 5 |
| [76] | augmentation | • | | | • | • | | | | | | • | | | | • | | C, C++ | ISSTA | 2016 | Yoshida | 3 |
| [52] | generation | • | | | | | | • | | | | | | | | | • | C | IST | 2017 | Patrick | 4 |
| [3] | augmentation | | • | | | • | | | | | • | | • | | | • | | Java | TAIC PART | 2006 | Apiwattanapong | 3 |
| [58] | augmentation | | • | | | • | | | | | • | | • | | | • | | Java | ASE | 2008 | Santelices | 3 |
| [21] | repairing refactoring | | • | | • | | | | | | • | • | | | | | | Java | ASE | 2009 | Daniel | 4 |
| [69] | augmentation | | • | | | • | | | | | | | | | • | | | Java | APSEC | 2009 | Xu | 3 |
| [55] | | | • | | | • | | | | | | | • | | | • | | C | ASE | 2010 | Qi | 4 |
| [65] | augmentation | | • | | | • | | | | | | | • | | | | • | Java | GECCO | 2010 | Xu | 3 |
| [68] | augmentation | | • | | | • | | | | | | | • | | • | | • | C | FSE | 2010 | Xu | 2 |
| [59] | augmentation | | • | | | • | | | | | • | | • | | | | | Java | ICST | 2011 | Santelices | 3 |
| [67] | augmentation | | • | | | • | | | | | | | • | | • | | • | C | ISSRE | 2011 | Xu | 3 |
| [42] | repairing adaptation | | • | | • | • | | | | • | | • | • | | | | | | Java | ICST | 2012 | Mirzaaghaei | 3 |
| [43] | repairing adaptation | | • | | • | • | | | | • | | • | • | | | | | | Java | SVTR | 2014 | Mirzaaghaei | 3 |
| [13] | | | • | | | • | | • | | | | | • | | | • | • | C | ESEC/FSE | 2013 | Böhme | 3 |
| [37] | | | • | | | • | | | | | | | • | | | • | • | C | ESEC/FSE | 2013 | Marinescu | 5 |
| [62] | augmentation | | • | | | • | | | | | • | | • | | | • | | Java | CSTVA | 2014 | Wang | 3 |
| [66] | augmentation | | • | | | • | | | | | | | • | | • | | • | C | STVR | 2015 | Xu | 3 |
| [48] | | | • | | | • | | | | | • | | • | | | • | | C | ICSE | 2016 | Palikareva | 4 |
| [82] | amplification | | | • | | | | • | | | | | • | • | | | | Java | ICSE | 2012 | Zhang | S |
| [83] | amplification | | | • | | | | • | | | | | • | • | | | | Java | TOSEM | 2014 | Zhang | |
| [35] | amplification | | | • | | | | • | | | | | • | | | | | CUDA | PLDI | 2012 | Leung | S |
| [19] | amplification | | | • | | | | • | | • | | | • | • | | | | Java | IST | 2015 | Cornu | 1 |
| [23] | amplification | | | • | | | | • | | | | | • | | | | | Java | ECOOP | 2015 | Fang | 1 |
| [81] | augmentation | | | • | | • | | • | | | • | | • | | | • | | Java | FSE | 2016 | Zhang | 3 |
| [26] | amplification | | | | • | | | | | | • | | | | | | | | ISSTA | 1993 | Hamlet | S |
| [64] | augmentation | | | | • | | | | | | | • | | • | | | | Java | ECOOP | 2006 | Xie | 5 |
| [33] | amplification | | | | • | | | | | | • | | | | • | | | C | ESEC/FSE | 2007 | Joshi | S |
| [45] | | | | | • | | | | | | • | • | | | | | | Java | ICST | 2009 | Mouelhi | 4 |
| [20] | enrichment | | | | • | | | | | | • | • | | | | | | Java | ISSTA | 2010 | Dallmeier | 4 |
| [17] | cross-checking | | | | • | | | | | | • | | | • | | | | Java | ICSE | 2014 | Carzaniga | 6 |
| [72] | purification | | | | • | | | | • | | | • | | | | | | Java | FSE | 2014 | Xuan | 5 |
| [70] | purification refactoring | | | | • | | | | | • | | • | | | | | | Java | IST | 2016 | Xuan | 1 |

47 papers included in the table have been published between 2003 and 2017. One paper was published back in 1993. Between years 2009 and 2016 the number of papers has been stable (mostly four or five per year). In 2014 two extensions to previous works have been published in addition to five original works, making it the year with most publications on the subject.

Figure 1 visualizes the snowballing process. Every node of the graph corresponds to a review paper. Seed papers are represented as filled rectangles to distinguish them from the rest. All nodes incorporated in the same iteration are clustered together. The edges shown in the graph correspond to the references we followed for the paper inclusion. Backward references are marked in green and labelled "B". For these edges, the origin node cites the target node. Forward references are marked in blue and labelled "F". For these edges, the origin node is cited by the target node.

### *7.2. Technical Aspects*

Most works include some form of test or application code analysis (26 and 21 contributions respectively). Notably, the majority of works that add new test cases also include a test code analysis phase. All papers that amplify the test suite with respect to a change also include an application analysis stage. Search-based heuristics and symbolic execution are used to a large extent (12 contributions each), while concolic execution and execution modification are the least used techniques (5 contributions each).

Java programs are the most targeted systems (30 contributions), followed by C programs (12 contributions). JavaScript applications have received very little attention in the area (only one row).

### *7.3. Tools for Test Amplification*

Most test case amplification papers discussed in this paper are experimental in nature, and are based on a prototype tool. For the field to mature, it is good if researchers can reproduce past results, and compare their new techniques against existing ones. To this extent, we feel that open-science in the form of publicly-available and usable research prototypes is of utmost importance.

With this in mind, we have surveyed not only the articles, but also the mentioned tools, if any. The protocol was as follows. First, we looked for a URL in the paper, pointing to a web page containing the code of the tool or experimental data. For each URL, one of the authors opened it in a browser between March 1st and March 31st 2018, to check that the page still exists and indeed contains experimental material.

Table 3 contains all valid URLs found. Overall, we have identified 17 valid open-science URLs. It may be considered as a low ratio, and we thus call for more open-science and reproducible research in the field of test amplification.

### *7.4. Open Questions for Future Research*

Most of the work discussed targets the unit test level, i.e., small tests that verify singles behaviors. Yet we do not see conceptual barriers to using them in acceptance tests of GUI tests such as Selenium.

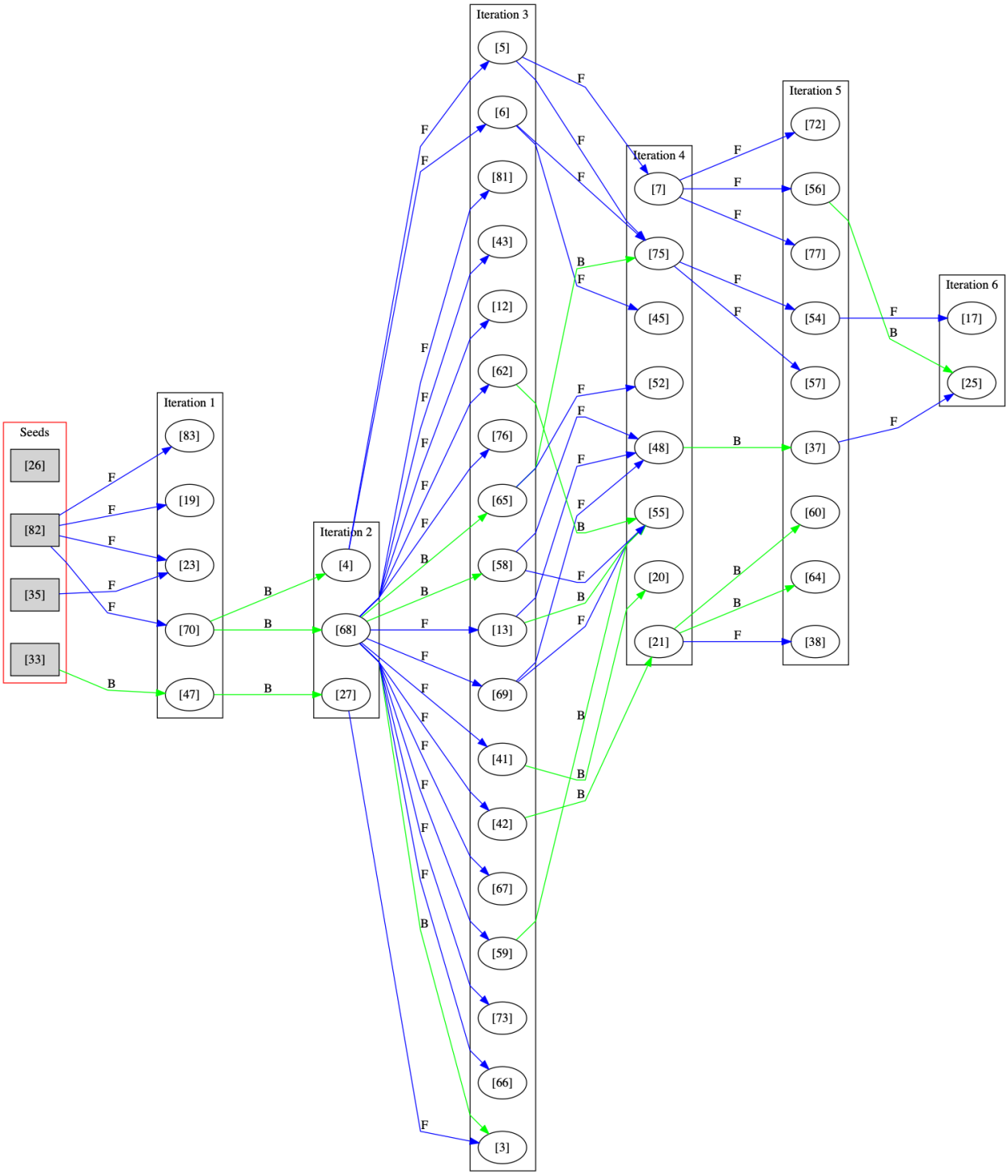


Figure 1: Visualization of the snowballing process. Each node corresponds to a paper included in the study. Seed papers are differentiated form the rest. Papers added in the same iteration are clustered together. F blue edges represent forward references. B represent backward references.

## 8. Threats to validity

Since conducting a survey is a largely manual task, most threats to validity relate to the possibility of researcher bias, and thus to the concern that other researchers might come to different results and conclusions. One general remedy that we adopted to counter this, is to work in a structure way, i.e., by starting from a small set of seed papers, use the citation graph to discover new papers.

In the following, we describe validity threats and discuss the manners in which we attempted to minimize their risk.

*Article selection.* Test amplification is a relatively new and narrow subject and we found that, as yet, there is no consensus on terminology. Therefore, we used a snowballing survey, which is less likely to be affected by the use of diverse terminologies. This approach is also immune for the issues of keyword based searches, which as has been observed by Brereton et al., can be problematic [16].

*Completeness.* We have addressed the threat of selection bias by utilizing the aforementioned snowballing approach. However, some related work could be missing either because they use a very original name for referring to test amplification or because it has not yet been cited by any of the seed papers that we used to start the snowballing effort.

*Article categorisation.* We have organised the papers in our survey in four categories, through an incremental analysis of the techniques and goals of each paper. The construction of the categorisation is subjective and may be difficult to reproduce. To minimize this risk, two authors performed the categorisation and had to reach consensus, both at the level of creating categories and assigning papers to categories.

## 9. Conclusion

We have studied the literature related to test amplification. This survey is the first that draws a comprehensive picture of the different engineering goals proposed in the literature for test amplification. In particular, we note that the goal of test amplification goes far beyond maximizing coverage only. We also give an overview of the different techniques used, which span a wide spectrum, from symbolic execution to random search and execution modification.

We believe that this study will help future PhD students and researchers entering this new field to understand more quickly and more deeply the intuitions, concepts and techniques used for test amplification. Finally, we note the lack of work that tries to compare "traditional" test generation (generating test cases from scratch), for which there is a myriad of papers, and test amplification (generating tests from existing tests). We think that sound and systematic experimental comparison of different test creation techniques would be a milestone for the nascent and emerging field of test amplification.

## Acknowledgement

This work has been partially supported by the EU Project STAMP ICT-16-10 No.731529.

## References


[1] Software-artifact infrastructure repository. `http://sir.unl.edu`. Accessed: 2017-05-17.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, pages 137–146. IEEE, 2006.

[4] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to .net components. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, ASE '02, pages 253–, Washington, DC, USA, 2002. IEEE Computer Society.

[5] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Automatic test cases optimization: a bacteriologic algorithm. *IEEE Software*, 22(2):76–82, Mar. 2005.

[6] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software, Testing, Verification & Reliability journal (STVR)*, 15(2):73–96, June 2005.

[7] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 82–91, 2006.

[8] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman. Developer testing in the IDE: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, 2019.

[9] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 179–190. ACM, 2015.

[10] M. Beller, G. Gousios, and A. Zaidman. How (much) do developers test? In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 559–562. IEEE Computer Society, 2015.

[11] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[12] R. Bloem, R. Koenighofer, F. Röck, and M. Tautschnig. Automating test-suite augmentation. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 67–72. IEEE, 2014.

[13] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 334–344. ACM, 2013.

[14] M. Böhme and A. Roychoudhury. Corebench: Sbohme2014corebenchtudying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 105–115. ACM, 2014.

[15] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571–583, 2007.

[16] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571–583, 2007.

[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*, ICSE 2014, pages 931–942, 2014.

[18] H. M. Cooper. *Synthesizing Research: A Guide for Literature Reviews*, volume 2. Sage, 1998.

[19] B. Cornu, L. Seinturier, and M. Monperrus. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Information and Software Technology*, 57:66–76, 2015.

[20] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM.

[21] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 433–444, 2009.

[22] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, 1999.

[23] L. Fang, L. Dou, and G. Xu. Perfblower: Quickly detecting memory-related performance problems via amplification. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[24] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 416–419. ACM, 2011.

[25] G. Fraser and A. Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 364–374. ACM, 2011.

[26] D. Hamlet and J. Voas. Faults on its sleeve: amplifying software reliability testing. *ACM SIGSOFT Software Engineeri ng Notes*, 18(3):89–98, 1993.

[27] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proc. of the Int. Conf. on Software Engineering (ICSE)*, pages 60–71, 2003.

[28] M. J. Harrold and A. Orso. Retesting software during development and maintenance. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 99–108, 2008.

[29] W. C. Hetzel. *The Complete Guide to Software Testing*. QED Information Sciences, Inc., Wellesley, MA, USA, 2nd edition, 1988.

[30] M. Hilton, J. Bell, and D. Marinov. A large-scale study of test coverage evolution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 53–63. ACM, 2018.

[31] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[32] S. Jalali and C. Wohlin. Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 29–38. ACM, 2012.

[33] P. Joshi, K. Sen, and M. Shlimovich. Predictive Testing: Amplifying the Effectiveness of Software Testing. In *Proc. of the ESEC/FSE: Companion Papers*, ESEC-FSE companion '07, pages 561–564, New York, NY, USA, 2007. ACM.

[34] B. Kitchenham. Procedures for performing systematic reviews. Technical report, Keele University, 2004.

[35] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner. Verifying gpu kernels by test amplification. *ACM SIGPLAN Notices*, 47(6):383–394, 2012.

[36] L. Madeyski. *Test-Driven Development: An Empirical Evaluation of Agile Practice*. Springer, 2010.

[37] P. D. Marinescu and C. Cadar. KATCH: high-coverage testing of software patches. page 235. ACM Press, 2013.

[38] M. R. Marri, S. Thummalapenta, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. Technical report, North Carolina State University, 2010.

[39] P. McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[40] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, 2006.

[41] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 67–78. ACM, 2014.

[42] M. Mirzaaghaei, F. Pastore, and M. Pezze. Supporting test suite evolution through test case adaptation. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 231–240. IEEE, 2012.

[43] M. Mirzaaghaei, F. Pastore, and M. Pezzè. Automatic test case evolution. *Software Testing, Verification and Reliability*, 24(5):386–411, 2014.

[44] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In T. Mens and S. Demeyer, editors, *Software Evolution*, pages 173–202. Springer, 2008.

[45] T. Mouelhi, Y. Le Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 171–180. IEEE, 2009.

[46] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[47] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European conference on Object-Oriented Programming*, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag, Springer Berlin Heidelberg.

[48] H. Palikareva, T. Kuchta, and C. Cadar. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1181–1192. ACM, 2016.

[49] F. Palomb and A. Zaidman. The smell of fear: On the relation between test smells and flaky tests.

[50] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE Computer Society, 2017.

[51] C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):339–353, 2009.

[52] M. Patrick and Y. Jia. Kd-art: Should we intensify or diversify tests to kill mutants? *Information and Software Technology*, 81:36–51, 2017.

[53] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson. Systematic Mapping Studies in Software Engineering. In *EASE*, volume 8, pages 68–77, 2008.

[54] M. Pezze, K. Rubinov, and J. Wuttke. Generating effective integration test cases from unit ones. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 11–20. IEEE, 2013.

[55] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 397–406, 2010.

[56] J. Rö$\beta$ler, G. Fraser, A. Zeller, and A. Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 309–319. ACM, 2012.

[57] J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, 2016.

[58] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *23rd IEEE/ACM International Conference on*, pages 218–227. IEEE, 2008.

[59] R. Santelices and M. J. Harrold. Applying aggressive propagation-based strategies for testing changes. In *IEEE Fourth International Conference on Software Testing, Verification and Validation*, pages 11–20. IEEE, 2011.

[60] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE software*, 23(4):38–47, 2006.

[61] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[62] H. Wang, X. Guan, Q. Zheng, T. Liu, C. Shen, and Z. Yang. Directed test suite augmentation via exploiting program dependency. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 1–6. ACM, 2014.

[63] C. Wohlin. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 38. ACM, 2014.

[64] T. Xie. Augmenting Automatically Generated Unit-test Suites with Regression Oracle Checking. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 380–403, 2006.

[65] Z. Xu, M. B. Cohen, and G. Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1365–1372. ACM, 2010.

[66] Z. Xu, Y. Kim, M. Kim, M. B. Cohen, and G. Rothermel. Directed test suite augmentation: an empirical investigation. *Software Testing, Verification and Reliability*, 25(2):77–114, 2015.

[67] Z. Xu, Y. Kim, M. Kim, and G. Rothermel. A hybrid directed test suite augmentation technique. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 150–159. IEEE, 2011.

[68] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 257–266. ACM, 2010.

[69] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Software Engineering Conference, 2009. APSEC'09. Asia-Pacific*, pages 406–413. IEEE, 2009.

[70] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus. B-Refactoring: Automatic Test Code Refactoring to Improve Dynamic Analysis. *Information and Software Technology*, 76:65–80, 2016.

[71] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[72] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63. ACM, 2014.

[73] J. Xuan, X. Xie, and M. Monperrus. Crash Reproduction via Test Case Mutation: Let Existing Test Cases Help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 910–913, New York, NY, USA, 2015. ACM.

[74] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[75] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22(3):171–201, 2012.

[76] H. Yoshida, S. Tokumoto, M. R. Prasad, I. Ghosh, and T. Uehara. FSX: Fine-grained Incremental Unit Test Generation for C/C++ Programs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, 2016.

[77] Z. Yu, C. Bai, and K.-Y. Cai. *Inf. Softw. Technol.*, 55(12):2076–2098, Dec. 2013.

[78] V. G. Yusifoğlu, Y. Amannejad, and A. B. Can. Software test-code engineering: A systematic mapping. *Information and Software Technology*, 58:123 – 147, 2015.

[79] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *First International Conference on Software Testing, Verification, and Validation (ICST)*, pages 220–229. IEEE Computer Society, 2008.

[80] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[81] J. Zhang, Y. Lou, L. Zhang, D. Hao, L. Zhang, and H. Mei. Isomorphic regression testing: Executing uncovered branches without test augmentation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 883–894, New York, NY, USA, 2016. ACM.

[82] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering*, pages 595–605. IEEE Press, 2012.

[83] P. Zhang and S. G. Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Trans. Softw. Eng. Methodol.*, 23(4):32:1–32:28, 2014.

Table 3: List of surveyed papers in which we have found a URL related to a tool

| Reference | URL | Observations |
|---|---|---|
| [1] | http://sir.unl.edu | This is a software repository. It is not a tool for amplification but it is a resource that could be used for amplification. |
| [7] | http://www.irisa.fr/triskell/results/Diagnosis/index.htm | The URL points only to results. |
| [14] | http://www.comp.nus.edu.sg/~release/corebench/ | The website also contains empirical results. |
| [17] | http://www.inf.usi.ch/phd/goffi/crosscheckingoracles/ | |
| [20] | https://www.st.cs.uni-saarland.de/models/tautoko/index.html | |
| [21] | http://mir.cs.illinois.edu/reassert/ | |
| [23] | https://bitbucket.org/fanglu/perfblower-public | There is no explicit url in the paper but a sentence saying that the tool is available in Bitbucket. With this information it was easy to find the URL. |
| [24] | http://www.evosuite.org/ | Additional materials included. |
| [38] | https://sites.google.com/site/asergrp/projects/putstudy | The website also contains empirical results. |
| [41] | https://github.com/saltlab/Testilizer | |
| [47] | http://groups.csail.mit.edu/pag/eclat/ | The website provides basic usage example. |
| [48] | https://srg.doc.ic.ac.uk/projects/shadow/ | The website also contains empirical results. |
| [54] | http://puremvc.org/ | The paper has been turned into a company. The provided url is the url of this company. |
| [56] | https://www.st.cs.uni-saarland.de/bugex/ | The url lives, but there is no way to download and try the tools. |
| [70] | https://github.com/Spirals-Team/banana-refactoring | |
| [71] | https://github.com/SpoonLabs/nopol | Still active. |
| [81] | https://github.com/sei-pku/Ison | |