STAMP

Deliverable D11

State of practice for unit testing and test assessment

Project Number	:	731529
Project Title	:	STAMP
Deliverable Type	:	Report

Deliverable Number	:	D11
Title of Deliverable	:	State of practice for unit testing and test assessment
Dissemination Level	:	Public
Contractual Delivery Date	:	M6 May, 31 2017
Contributing WPs	:	WP 1
Editor(s)	:	Caroline Landry, INRIA Oscar Luis Vera Perez, INRIA
Author(s)	:	ActiveEon Atos INRIA OW2 XWiki

Abstract

Revision History

Version	Type of Change	Author(s)
0.01	initial setup and partners contributions	Caroline Landry, INRIA
0.10	First draft	Oscar Luis Vera Perez, INRIA
1.00	Final draft after review	Oscar Luis Vera Perez, INRIA

Contents

Acronyms

EC	European Commission

Chapter 1

Introduction

Software testing is the art of analyzing and exercising software in order to assess its dependability. Automatic test generation is a traditional area in software testing: the goal is to generate tests according to a specific test criterion. Traditionally, test generation assumes that no test exists at all, or that they are either too few or of too low quality for being considered useful in the generation process.

However, with the progress of agile development methodologies, which advocate test-driven development, it is now the state-of-the-art to have programs accompanied with a strong test suite. This test suite is usually large, and has been written on top of a lot of human intelligence and domain knowledge. Developers spend a lot of time in writing the tests, so that those tests exercise interesting cases (including corner cases), and so that a good oracle verifies as much as possible the program behavior.

The wide presence of strong manually written tests has triggered a new thread of research that consists of leveraging the value of existing tests to achieve a specific engineering goal. This is what we call in this document "test amplification". We note that the literature uses not only the term "amplification" but also other expressions such as "augmentation". There is no consensus on the differences between all terms. To our opinion, this divergence is natural in new fields. It is normal that the involved community tinkers a bit before converging to a specific expression. We use the term "amplification", because it well conveys the idea that a machine or an algorithm improves an existing thing coming from humans, as a sound amplifier amplifies the voice of pop singer Shakira.

This deliverable presents first a survey of this emerging and novel field in software testing, the goal of this survey being to strengthen the foundations of test amplifications and foster further work in this promising research direction. The document also includes sections that describe the methodologies and tools employed by the partners of the project in practical unit and integration testing.

Novelty

There are a number of notable surveys in software testing. However none of them is dedicated to test amplification. For instance, we refer to Edvardsson's et al's [11] and McMinn et al's [18] articles for a survey on test generation. Yoo and Harman have structrured the work on test minimization, selection and prioritization [46] . In the prolific literarure on symbolic execution for testing, we refer the reader to the survey of Păsăreanu and Visser [24]. 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, however, as meant by Van Deursen et al [34], it is different in nature from test amplification as considered in this paper, because classical test refactoring is not fully automated. On the contrary, test amplification is meant to be fully automated, as other technical amplifications such as as sound amplification. Harrold et al. [15] 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.

Methodology

Before starting the survey, we scope test amplification as follows. This scope naturally defines an inclusion criterion, and we will discuss all notable papers that fit into this scope.

> In this paper, test amplification means improving the efficiency of a test suite with respect to an engineering goal.

The most commonly considered engineering goals are: maximize a test criterion (mostly coverage); improve observability; assess properties of the test suite under study; assess properties of the application under study and improve the applicability of the test suite with respect to a certain usage.

To survey the literature a systematic methodology was followed. A first set of papers based on the knowledge of the literature from most experimented partners was devised. Then, the citation graph was systematically followed: backwards for finding the first papers mentioning this idea (such as [14]) and forwards in order to find the most recent contributions in this area. The main goal is to be inclusive, papers are included based on the novelty and significance of the idea they contain and not filtered by specific conferences or journals.

Finally we classified and ordered the considered papers in categories. This categorization is not exclusive, and other categorization may be found as well. However, it well conveys the main threads of research in test amplification.

The document is organized as follows:

Chapter 2 presents techniques that synthesize new tests from existing ones.

Chapter 3 focuses on the works that synthesize new tests dedicated to a specific change in the application code (in particular a specific commit).

Chapter 4 discusses the less researched yet powerful idea of modifying the execution of tests for amplifying.

Chapter 5 is about the modification of existing tests.

Chapter 6 recapitulates the main techniques and goals that we have found in the literature.

Chapters 7, 8, 9 and 10 contain descriptions of the methodologies and tools employed by the industrial partners.

Chapter 11 concludes this report.

Chapter 2

Addition of new tests synthesized from existing ones

Here we address the most intuitive form of test amplification: considering an existing test suite, automatically generate variants of the existing test cases, and add these variants into the test suite. This is one kind of test amplification, that we call AMP_{add}.

Definition: Test amplification technique AMP_{add} consists of creating new tests from existing ones so as to improve an engineering goal. The most commonly used engineering goal is to improve coverage according to a coverage criterion.

Coverage or Mutation Score Improvement

To improve the mutation score of an existing test suite, Baudry *et al.* [3] [2] propose to generate variants of existing test cases through the application of specific transformations (e.g., modify literal values in method calls). 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.

Tillmann and Schulte [33] describe a technique that can generalize existing unit tests into parametrized 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. The problem of generalizing unit tests into parametrized unit tests is also studied by Thummalapenta et al. [17]. Their empirical study shows that unit test generalization can be achieved with feasible effort, and can bring the benefits of additional code coverage.

Using two open source programs as the study subjects, Smith and Williams [31] empirically evaluate the usefulness of mutation analysis in improving an existing test suite. They execute the existing test cases against a set of generated mutants and for each mutant that is not killed, a new test case is written intentionally to kill it and kill only it. Their results reveal that a majority of mutation operators is useful for producing new tests, and the focused effort on increasing mutation score leads to increase in line and branch coverage. The same authors later conduct another study [32] to further confirm this finding, and the importance of choosing appropriate mutation tool and operators for using mutation analysis to produce new test cases is explicitly pointed out in this new study.

To improve the cost efficiency of the test generation process, Yoo and Harman [47] 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.

To maximize code coverage, Bloem et al. [5] propose an approach that alters existing tests to get new tests that enter new terrain. 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.

Fault Detection Capability Improvement

Starting with a source of test cases, Harder et al. [14] 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.

Pezze et al. [26] 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 has the advantage of gaining domain knowledge of the developers. Instead, 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. [19] 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.

Oracle Improvement

Pacheco and Ernst implement a tool called Eclat [23], 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 ones that are most likely to reveal faults. Finally, Eclat adds an oracle for each remaining test input from the operational model automatically. It has been shown that Eclat can generate inputs that reveal real errors in Java classes.

Given some test generation techniques just generate sequences of method calls but do not contain oracles for these method calls, Fraser and Zeller [13] 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.

Debugging Effectiveness Improvement

Baudry *et al.* [4] establish the link between testing and fault localization. In particular, they 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 test-for-diagnosis criterion can be satisfied.

Röβler et al. [28] propose BugEx approach, which leverages test case generation to systematically isolate failure causes. The approach takes a single failing test as input and begins with 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 are 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.

Yu et al. [49] 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 data augmentation technique that is capable of generating test suites with excellent 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 increases the chance of executing the specific path during crash reproduction, Xuan et al. [45] 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 crash.

Instead of operating at the granularity of complete test cases, Yoshida et al. [48] 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.

Recapitulation

Main achievements: Compared with generating tests from scratch, using existing test cases as a start point to generate new test cases can make the test generation process more targeted and cost-effective. On the one hand, 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 coverage 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. For instance, there exist several pieces of work that apply mutations to the existing tests and the generated tests have been shown to be especially effective for debugging.

Main challenges: While existing tests provide a good starting point, there still exist some difficulties in how to better make use of the information contained in the existing tests. First, the number of new tests synthesized from existing ones can be large sometimes and hence an effective strategy should be used to select tests useful for the specific engineering goal. Second, the synthesized test can be invalid occasionally. For example, while it is relatively easy to synthesize new test inputs from the existing tests, it can be hard to add oracles for the synthesized new inputs sometimes. Finally, for some specific engineering goals, it is difficult to accurately measure how each test perform with respect to those goals.

Chapter 3

Synthesis of new tests with respect to software changes

Software applications are not tested at once. They are rather tested incrementally, along with the natural evolution of the code base: new tests are added together with a change or a commit, for instance to verify 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 according to a change. This can be seen as a specialized form AMP_{add}, which considers a specific change, in addition to the existing test suite, to guide the amplification. We call this form of test amplification AMP_{change}

Definition: Test amplification technique AMP_{change} consists of 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 in finding testing conditions to exercise the portions of code that exhibit changes. A third subsection relates other promising works in the same area.

Search based vs. concolic based approaches

In their work, Xu et al. [41] focus on the scenario where a program has evolved into a new version through code changes in development. They consider test augmentation 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 not covered by the test suite. In 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. Targeting paths related to uncovered branches prevents a full concolic execution, which improves the performance of the augmentation process. They applied their technique to 22 versions of a small arithmetic program from the SIR 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. [37] 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 efficacy and efficiency. The authors conclude that considering all tests achieve 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 [40]. 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 [39]. 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 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 theorise about possible strategies for combining both individual approach to overcome their respective weaknesses and exploit their best features.

A revised and extended version of the experiments performed to validate the proposals made by this group is given in [38].

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 only in when particular conditions are met by the input. The following papers address the problem of finding concrete input conditions that not just can execute the changed code, but propagate the effects of this change to an observable point that could be the output of involved test cases. 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. [1] 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 separated generation process and is not able to handle changes to several statements unless the changed statements are unrelated.

Santelices et al. [29] 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 don't compute state requirements for changes affected by others. Empirical evidence is provided regarding the capacity of this method over traditional coverage criteria to assess test conditions for program changes.

In a third paper [30] the 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. This approach also considers multiple program changes. Removing the need of symbolic execution leads to a less expensive method. They claim that propagation-based testing strategies are superior to coverage-based in the presence of evolving software.

Other approaches

Other authors have also explored test suite augmentation for evolving programs with propagation-based approaches. Qui et al. [27] propose a method to add new test cases to an existing test suite ensuring that effects of changes in the new program version are observed in the test output. The technique consists in a two step symbolic execution. First, they explore the paths towards the program change 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.

Wang et al. [35] exploit existing test cases to generate new ones, which 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 conditions 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 3 Java programs 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.

In a different direction, Bohme et al. [6] 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 the *GNU Coreutils* toolset. They were able to find 5 unknown errors in addition to previously reported issues.

Recapitulation

Main achievements: Symbolic and concolic execution are recurrently used among this group of papers. Both have been successfully combined with other techniques in order to generate test cases that reach changed or evolved parts of a program and therefore increasing coverage in the new program version. The use of data and control dependency analysis along with different notions of distance allows to narrow the search space, guides the test case generation and has a positive impact in the computation cost.

Main challenges: Despite the progress made in the area, a number of challenges remain open. The use of symbolic and concolic execution has proven to be effective in test input generation targeting program changes, but these two techniques are expensive in terms of computation resources. New and more efficient ways for exploring input requirements that could exercise program changes or new uncovered parts are needed. One of the previously cited papers removes the use symbolic execution by observing the program behavior during test execution but they don't generate test cases. Most works on this area consider a single change in a single statement. should consider not only the effect of one separated change but also the effect of all changes as a group and even the interaction between them. Only one paper makes a proposal in this direction.

Chapter 4

Tests Execution Modification

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

Definition: Test amplification technique AMP_{exec} consists of modifying the test execution process or the test harness in order to maximize the knowledge gained from the testing process.

Zhang and Elbaum [51] 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 amplify the test suite triggering unexpected exceptions in sequences of API calls. Issues are detected when there is an abnormal termination of the program or abnormal execution time while running the tests. The approach is shown to be cost-effective and able to detect real-life problems in 5 Android applications.

Cornu et al. [8] work in the same line of evaluating exception handling. 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 a formal specification 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 approach is evaluated in 9 Java real-life open source projects.

Fang et al. [12] develop a performance testing system named *Perfblower*, able to detect and diagnose memory issues by observing the execution of a set of test cases. The system includes an instrumentation specific language designed that developers can use to describe the symptoms of ill memory usage patterns in code. Based on the provided descriptions, the tool evaluates the impact of detected problems. One thing to notice is that, the term "amplification" is used with a different meaning, specific to their implementation details. The approach is evaluated in 13 Java real-life projects. The tool was able to find real memory issues and reduce the number of false positives reported by other similar tools.

Zhang et al. [50] 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 behaviours 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* 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*.

Recapitulation

Main achievements: Proposals in this line of work usually provide cost-effective approaches to observe a program during test execution and detect possible faults. Instrumenting the original program code to place observations at certain points or mocking resources to monitor API calls to explore unexpected scenarios adds no prohibitive overhead to regular test execution and provide 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: A small number papers are include in this section. This could mean that more works that propose ways to enhance the testing process during test execution need to be explored as a counterpart of new test generation approaches. Their efficiency is a pro to consider for this matter.

Chapter 5

Existing Tests Modification

In testing, it is up to the developer to arrange the tests. The main testing infrastructure such Junit in Java does not impose anything on the tests, such as the number of statements in a test, or the cohesion of test assertions. There is work on modifying existing tests with respect to a certain engineering goal.

Definition: Test amplifying technique AMP_{mod} refers to modifying the body of existing test methods. Differently from AMP_{add}, it is not about adding new test methods or new tests classes.

Input Space Exploration

Dallmeier et al. [9] 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 execution states than the original test suite in order to improve the quality of models reverse engineered from the code. They evaluate their tool, *TAUTOKO*, on 7 java classes and show it able to produce a richer typestate. Typestate is a finite state automaton which encodes legal usages of a class under test. It is use to run a typestate verification. This verification discovers illegal transition. The richer is the input typestate, more the verification reports defects in the code.

Oracle Improvement

Xie [36] amplifies object-oriented unit tests. The technique 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 behaviors of the given method, which depends on the state of the receiver and of arguments at the begin of the invocation. The approach, named *Orstra*, consists of the instrumentation of the bytecode, running the test suite to collect state of objects. Then, he uses instrumentation of the byte code to retrieve the state of the program with method calls sequence. Then it generates assertions, using observers (pure method with a non-void return type, *e.g.* $toString()$) and the collect values as oracle. He evaluated *Orstra* on 11 Java classes and their tests, and shows that it is able to increase the fault-detection capability of the test suite.

Carzaniga *et al* [7] generate generic oracles, that is to say a generic procedure to assert the behavior of the system under test. To do so, they exploit the redundancy of software. Redundancy of software happens when the system can perform the same actions through different executions, either with different code or with the same code but with different input parameters or in different contexts. They devised the cross-checking oracles, which compare the outcome of the execution of an original method call and an equivalent code. Such oracle use a generic equivalence check on the results and the state of the target object. If there is an inconsistency, the oracle reports it, otherwise, the checking continue. This oracles are added to an existing test suite with the aspect-oriented programming. They evaluate the approach on specific classes of 3 projects. They show that the approach can slightly increase (+6% overall) the mutation of score of a manual test suite.

Joshi et al. [16] try to amplify the effectiveness of testing the program on the provided test by executing both concretely and symbolically. Along this double execution, for every conditional statement executed by the concrete execution, the symbolic execution generate symbolic constraints over the input variables. At the execution of an assertion, the symbolic execute invokes a theorem prover to check the assertion is verified, according to the constraints encountered. If the assertion is not guaranteed, a violation of the behavior is reported. They evaluated their approach on 5 C programs. They are able to detect buffer overflow but it needs optimization because of the huge overhead that the instrumentation add.

Mouelhi *et al.* [22] enhance tests oracle for access control logic, also called Policy Decision Point (PDP). This is done in 3 steps: Select test cases that execute PDP, map each of test cases to a specific PDP 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 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. They evaluate the approach on 3 java projects. Students generated manually test cases to be compared with the results of the approach. The generation saves a lot of time (from 32 hours to 5 minutes) but required that the PDP send specific messages when an access is denied.

Purification

Xuan et al. [42] propose a technique to split existing tests into smaller parts in order to “purify” test cases. For instance, an impure test case executes both branch *then* and *else* of the same if/then/else statement in code. The result of B-refactoring on this example is to have two different test cases that executes one branch only in isolation. They evaluate B-refactoring on 5 open-source projects, and show that it increases the purity of test cases. They the technique on 5 projects and show that individual elements are purer after applying B-refactoring: 66% for if statement and 11% for try statement. They also improve the effectiveness of program repair (Nopol [43]) by applying B-refactoring.

Xuan *et al* [44] 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 case and remove all statement that are not related to the assertion. Then, they refine the rank of the pure test cases with an existing test cases. They evaluated the test purification on 6 open-source java project, over 1800 seeded bugs and compare their results with 6 mature techniques. They show that they improve the fault localization on 18 to 43% of faults.

Test Case Repair

Daniel *et al* [10] devise *ReAssert* to repair automatically test cases. *ReAssert* follows 5 steps: records 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 and choose the repair strategy depending the structure the code and the recorded value. Finally, ReAssert re-compiles the code changes and repeats all steps until no more assertion fail. They evaluated with controlled user study. *ReAssert* could repair 98% (131 of 135) of failures caused by the participant code changes.

Mirzaaghaei *et al* [20] devise *TCA*, an automatic "repair" tools for unit test cases. According to a modification in the source code, *TCA* supports 4 repairs, *i.e.* changes in the test code as follows. First, “signature changes” of test methods, *i.e.* change of parameters of a method and/or the returned type value. Second, with "Test class hierarchy", when a developers create and extend new classes in the source code, *TCA* is able to generate test cases for the new class, based on test of the extended class. *TCA* generates test cases for overloading and overriding of methods, reusing existing test cases. Their implementation support only the Signature changes and Test class hierarchy. They evaluate the former technique on 9 projects of the Apache foundation. On average, for each project, *TCA* can repair the compilation errors for the 45% of the changes that lead to compilation errors. For the latter, can generate test cases for 60% of the classes of each of the 5 selected project, on average.

Recapitulation

Main achievements: AMP_{mod} is not commonly used to explore the input space of test. In the other hand, oracle improvement has been well explored, for instance Cross-checking oracle allows to increase (+6%) the mutation-score of a hand-written test suite. However, 2 others engineering goals has been well studied. The purification of test cases for if statement reaches a well state: 66%. The purification of test cases provide a better fault localization than others mature techniques (such Ochiai or Tarentula). And last but not least, Test case repair allows to repair test case for multiple problem, such as signature changes over large scale experimentation.

Main challenges: A first step to explore the input space by applying AMP_{mod} has to be done. Although the Cross-checking oracle seems to be effective, its evaluation remains very weak: they have been lead on few java classes (5-11). A larger evaluation is attended to validate results and effectiveness of techniques. In the other hand, test case purification has been well evaluated, but only the purification of test cases of if statement is well developed, the purification of test cases of try statement is very low (11%), it is one of the challenges of improving test purification. Finally, test case repair has 2 problem for now, The first is *ReAssert* is not fully automated. The second is the repair is consider good if there is no more compilation error. This is a strong assumption and can be not always the case. The validation of automatic-repair is a hard problem.

Chapter 6

Analysis

This section provides a brief overall analysis of existing work in the field.

Recapitulation on Engineering Goals

In the reviewed works, the test suite amplification process pursues a specific engineering goal. Table 6.1 shows the main goals we have identified and gathers the notable references for each category. Many papers describe techniques that try to maximize or increment the coverage that an original test suite has. Those are shown in row "maximize coverage". Another group of works aims to improve the observability of the test suite in the sense of providing additional information apart from the fact that a particular test case has failed or succeeded. This second group is listed in "improve observability". A few works directly target specific properties of the system under test such as robustness in exception handling or memory performance. Another small group do the same for the test suite. Those two groups are shown as "asses properties of the application under study" and "asses properties of the test suite under study" respectively. A last group contains papers proposing ways to enhance the applicability of a test suite with respect to some usage. Given the broad scope of the papers included in this category a subdivision is proposed. Some of the techniques considered would aim to reproduce a runtime crash ("crash reproduction"), detect the presence of faults ("fault detection capability"), locate faults in the program under test ("fault localization") and even try to propose a fix that could be applied ("repair").

Table 6.1: Overview of the different goals of test suite amplification. Only most the notable references are given.

Engineering goal	Works
maximize coverage	[14, 33, 41, 37, 39, 47, 19, 48, 50]
improve observability	[23, 36, 16, 9, 13, 30, 51, 7, 8, 12, 25]
assess properties of the test suite under study	[31, 32]
assess properties of the application under study	[51, 26, 8]
improve the applicability of the test suite wrt a usage	
crash reproduction	[45]
fault detection capability	[3, 50]
fault localization	[4, 44, 49, 42]
repair	[10, 20, 21]

Recapitulation on Technical Approaches

There are several different techniques to achieve test amplification. We conduct an analysis of the underlying technique used in each reviewed paper and classify the used techniques into 6 main different categories. Table 6.2 shows this classification in detail, with the first column showing the category and the second column showing the notable references that fall into each category.

The first category of technique is called test code analysis, which directly makes use of the characteristics of existing test code to achieve test amplification. Typically, this category of technique uses the structure or components of the existing test code as a starting point. The second category of technique is called application code analysis, which contains most references in this review. Another category of technique with many references in this review makes use of symbolic execution to amplify existing tests. Since symbolic execution facilitates the identification of path conditions and program states, this category of techniques generally generally aim at improving the efficiency of amplification. Meanwhile, a variant of symbolic execution named concolic execution is also used by some works to further improve the efficiency of test amplification. Finally, there exist some works that make use of search based heuristics such as genetic programming to amplify existing tests. Basically, the aim of using search based heuristics is also making the test amplification process more cost-efficient.

Test amplification can be achieved through the analysis of both the test and application code. In particular, recent advances in testing techniques such as symbolic execution and search based heuristics can make the amplification process more cost-efficient.

Table 6.2: Overview of the different ways of performing test suite amplification.

Category of techniques	Works
test code analysis	[10][20][7][26][49] [45][47] [36]
application code analysis	[30][42][36][31] [32] [14] [13] [12] [23][25][3][4]
execution modification	[51] [8] [50][44][9][10][20][21][19]
use of concolic execution	[41] [39]
use of symbolic execution	[1] [29] [27] [35] [6][16][33][48]
use of search based heuristics	[37] [39]

Chapter 7

Unit testing methodology at ActiveEon

As the previous sections reviewed the state of the art of test suite amplification in scientific papers, this section starts a review on the tools, metrics and methodologies followed by the project partners in practical unit and integration testing. This section focuses on the practices followed by ActiveEon.

Tools related to unit-testing

The main tools used by ActiveEon are :

- JUnit as base framework for testing.
- Hamcrest to compare the results to expected values.
- Mockito to mock out-of-scope parts to better isolate the code under test.
- ObjectFaker in order to build fake objects easily for testing purpose.
- MockServer which is a Java DSL for easy mocking REST services.
- DBUnit for test queries to the database.
- Rest-assured which is a java DSL for easy testing REST services.
- Gradle to manage the test suites. (test, integrationTest, functionalTest, regressionTest)

The consistency of these tools is ensured by the use of the same micro-service template (*ow2-proactive/yamt*) for a new micro-services.

Metrics

For the pull-request test suit and the system-test suit, the metrics are :

- The test duration
- The failed tests number
- The skipped tests number
- The succeeded tests number
- The total executed tests number
- For each metrics, the differences with the previous build.
- The stability of the build

Sonar metrics are :

- The coverage
- The new code coverage since previous version

- The skipped unit tests
- The unit test errors
- The unit test failures

Pipeline

Tests description

In the pipeline, unit tests, functional tests, sonar tests and system tests are automatic.

Unit test :

- Test a small piece of code
- Protect from modifications, regression test
- Identify quickly and at early stage what is wrong
- Simple entry point to work on a small piece of code

Functional Test :

- Communication between modules
- Individual modules tested as a group
- Test that modules and classes are able to talk to each other as expected

System-tests :

- Communication between components
- Test Protocols
- Test Database connections

Sonar test enables to ensure that code quality doesn't go under a predefined level configured by the sonar metrics.

For the manual testing developers should write a review, the demo tests and the manual tests:

- The developers review : It's a code review and comments.
- The demo test : It's the demonstration of the new feature on a test environment.
- The manual test : It ensures that the main feature still work on the new product version.

Tests by environment

In order to ensure the quality the tests are executed in the pipeline one different environment :

The developer local machine, the jenkins server and the trydev machine which is a test environment.

Locally Before creating a new pull request the following test should be executed :

- Unit tests
- Functional tests

Jenkins server On a pull request the following steps are done in order to ensure the pull request quality :

- Unit tests
- Functional tests
- Sonar quality assurance
- Two developers review (2 people read the pull requests and give their feedback about the pull request)

Trydev machine When a pull request is merged the following test are executed :

- System-test
- Demo test (each new feature is demonstrated to other developer)
- Manual test (ensure that the main feature still works)

The pipeline is really close to the flow suggested by github : *https://guides.github.com/*

Future

Performance tests are currently under development and will be tested on the trydev machine using the library JMeter.

Chapter 8

Survey techniques tools for code monitoring (Atos)

This section reports a survey on the state of practice on different technologies and tools for code unit testing, code coverage, static code analysis - including code quality and style checking -, detection of bugs and code smells, software profiling, and the way these tools are connected within the DevOps tool-chain, particularly during Continuous Integration (CI) and Continuous Delivery (CD). This survey is based on Atos ARI[1] current practices and experience, thus it is neither complete nor exhaustive.

Popular **software testing** (including unit, regression, system, integration, etc testing) tools are available for different programming languages and software types. In the Java realm, *JUnit*[2] is one of the most popular among developers for unit testing and desktop/backend system testing. It supports the definition of individual tests, aggregations in suites, test assertions, fixtures for sharing common test data, and assorted test runners. It is integrated with common build management tools (i.e. Maven, Spring, etc.), and IDE (i.e. Eclipse, IntelliJ). *TestNG*[3], another popular test framework, inspired by JUnit, supports additional features, such as additional annotations, safe multithreading, data-driven testing, etc., and it is also supported by mentioned build management tools and IDEs. *HttpUnit*[4] supports the testing of Web applications, as tested by a browser, supporting features as form submission, Javascript execution, http authentication, cookies and automatic page redirection. Page responses can be inspected for assertion. *Jasmine*[5] is a behaviour-driven JavaScript testing framework. It supports the execution of both front-end (using the Jasmine-JQuery extension) and Node.js tests. It supports the specification of individual tests and suites. It has a rich set of built-in matchers[6]. *Selenium*[7] is web application automation framework for testing, including components such as the Selenium IDE, the client API, the Remote Control or the WebDriver. It can be used to create regression testing tests and suites. It also generates scripts for bug reproduction and automation-aided exploratory testing. Test can be coded in different programming languages, including Java, Groovy, Scala and others, or by recording (for playing back) browser interaction.

Jacoco[8] is a Java **code coverage** library and Eclipse plugin. Code tests run from the IDE[9] are analysed for code coverage and the results are summarized and highlighted in the code editor. It

[1] Atos Research & Innovation (ARI) is the Atos department involved in STAMP project

[2] http://junit.org/

[3] http://testng.org/

[4] http://httpunit.sourceforge.net/

[5] https://jasmine.github.io/

[6] Matchers are similar to assertions in other frameworks such as JUnit

[7] http://www.seleniumhq.org/

[8] http://www.eclemma.org/jacoco/

[9] Using JUnit for example

provides coverage analysis of instructions, branches, lines, methods, types and McCabe cyclomatic[10] complexity. As it is based on Java byte code, it supports other JVM languages, such as Groovy or Scala. *Cobertura*[11] is another code coverage tool, based on jcoverage[12] library. As a code coverage tool, it calculates the percentage of code assessed by tests, therefore, aiding on identifying the portions of code not tested. Cobertura reports, per package, line and branch coverage and the cyclomatic complexity.

SonarQube is a **code quality** suite, helping developers to produce cleaner and bug-free code. It reports different code quality audits, including technical debt, code smells, bugs and vulnerabilities, code coverage and duplications, and so on. It supports different programming languages, including Java, JavaScript, Python, etc. *CodePro Analytix*[13] is another code quality plugin for Eclipse, used to detect quality issues and security vulnerabilities within the Java code. It includes over 1200+ audit rules (including 225 security rules) for static code analysis, supporting detection, repair, reporting and filtering. It also supports code coverage, duplicate and dead code detection, code metrics and so on. *CAST*[14] is another code quality tool used by Atos, which offers a complete software quality analysis for different dimensions, including business and technical. In the business dimension, it measures features such as security, changeability, transferability, etc. In the Technical dimension, it measures features such as efficiency, secure coding, programming practices, etc. Multiple-view reporting is quite complete, including code metrics, statistics about critical quality violations, technical debt,

VisualVM[15] is a Java **software profiling** tool that monitors and troubleshoots Java applications. In particular, Atos have been using VisualVM to monitor Java performance and memory, aiming at detecting performance bottlenecks and memory leaks. VisualVM reports on CPU usage, GC activity, heap memory, loaded classes, running threads, loading and browsing heap dumps (for instance upon the occurrence of OutOfMemoryErrors) or core dumps (for crashed Java processes).

Time ago, Atos also used Eclipse *TPTP*[16] plugin, but it was discontinued in latest Eclipse 4.X versions. It was quite convenient for Java profiling, as it was seamlessly integrated within the Eclipse JDT, as well with Eclipse WTP, for profiling Web applications deployed within J2EE containers (i.e. Tomcat). It has been replaced by VisualVM.

Some of these tools are integrated within the DevOps tool-chain in some Atos ARI projects, particularly during the CI/CD phases. JUnit/TestNG-based testing is integrated with build management tools such as Ant, Maven or Gradle. This building process is integrated in CI using Jenkins, as described in above figure that describes the recommended Atos ARI development methodology[17]. As described in the figure, code quality tools such as SonarQube is used to obtained the QA metrics of committed code before a merge request (e.g. pull request for GitHub) is accepted. This acceptance also requires successful pass of all tests.

[10] https://en.wikipedia.org/wiki/Cyclomatic_complexity

[11] http://cobertura.github.io/cobertura/

[12] https://java-source.net/open-source/code-coverage/jcoverage-gpl

[13] https://marketplace.eclipse.org/content/codepro-analytix

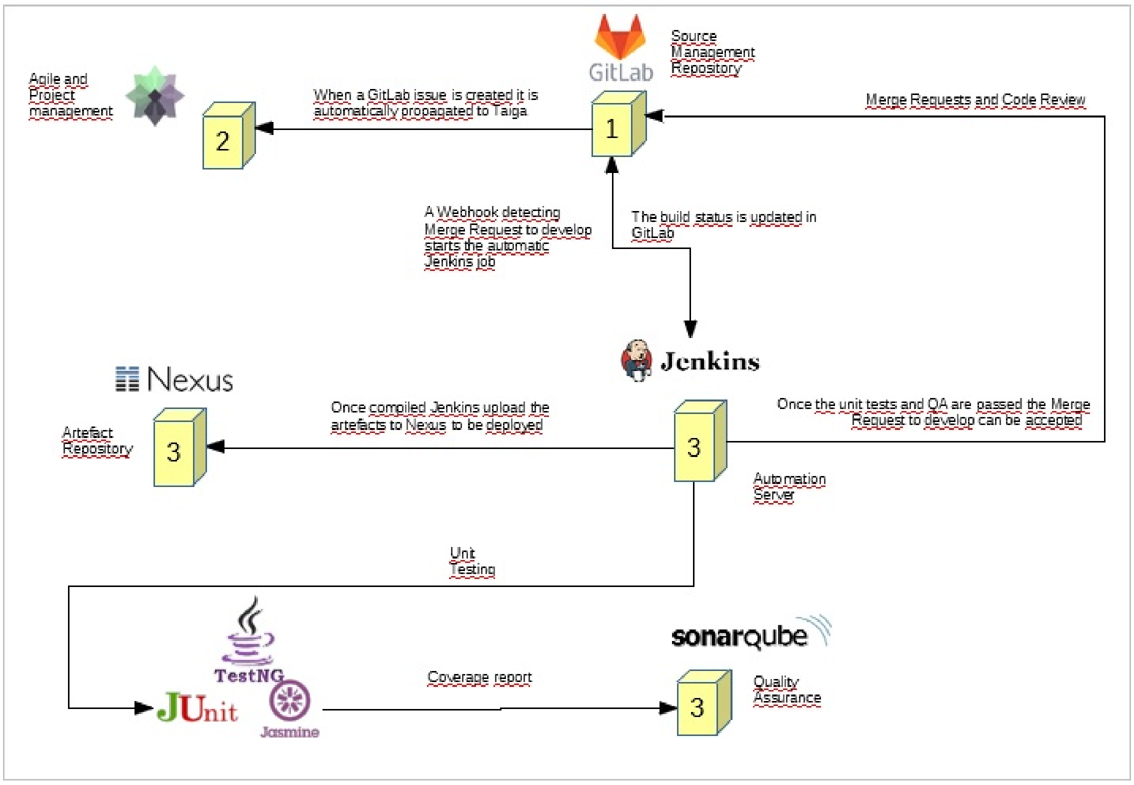
[14] http://www.castsoftware.com/

[15] https://visualvm.github.io/

[16] https://projects.eclipse.org/projects/tptp.platform

[17] This methodology is not completely adopted by all projects developed by Atos ARI, and in particular by the use cases managed by Atos team in the STAMP project, but it will be progressively adopted.

Figure 8.1: Common CI DevOps lifecycle in Atos ARI


GitLab
Source Management Repository
Agile and Project management
2
1
When a GitLab issue is created it is automatically propagated to Taiga
Merge Requests and Code Review
A Webhook detecting Merge Request to develop starts the automatic Jenkins job
The build status is updated in GitLab
Jenkins
Nexus
Artefact Repository
3
Once compiled Jenkins upload the artefacts to Nexus to be deployed
3
Once the unit tests and QA are passed the Merge Request to develop can be accepted
Automation Server
Unit Testing
TestNG
JUnit
Jasmine
Coverage report
sonarqube
3
Quality Assurance

Chapter 9

Unit Testing OW2 Survey

Introduction

The OW2 code base consists of approximately 25 mature projects and 30 projects in incubation. A survey has been conducted among 12 project leaders on their unit testing practices. The projects cover a wide range of programming languages and technologies. This document presents a summary of the received answers.

The surveyed projects are the following:

- AuthZForce, an Attribute-Based Access Control (ABAC) framework, compliant with XACML standard v3.0.
- CLIF, a multi-protocol, extensible load testing platform, including monitoring facilities, with optional integration to Eclipse IDE and Jenkins CI server.
- DocDoku, a Product Lifecycle Management solution.
- Erocci, an Erlang framework for bulding OCCI-compliant REST APIs.
- FusionDirectory, a interface for the management of LDAP servers.
- Hammr, a tool for creating machine images for different environments from a single configuration file.
- LemonLDAP::NG, a modular WebSSO (Single Sign On) based on Apache::Session modules.
- SAT4J, a library aiming at providing a simple and efficient open source library of SAT solvers in Java.
- Scarbo, an SOA ready, SCA powered Eclipse-based BPM solution.
- SeedStack, a lean development stack.
- SystemDesigner, a web-based IDE for designing JavaScript applications driven by the model.
- WebLab, a generic framework to crawl and index multimedia web documents.

Survey summary

The survey has consisted in submitting 25 questions to the project leaders. The paragraphs below synthetises the received answers.

While all the surveyed project write unit tests to test their code, only 4 out of 12 write parameterized tests (AuthzForce, SAT4J, FusionDirectory and LemonLDAP::NG), in different proportions: AuthzForce write almost only parameterized tests (99%), FusionDirectory 20% and Scarbo, SAT4J and LemonLDAP::NG only a few percents.

With respect to the framework used for running unit tests, the most popular framework among the projects is JUnit. DocDoku uses in addition Mockito and Jasmine. FusionDirectory uses Selenium for running user interface tests. Several projects use tools complementary to JUnit: AuthzForce uses TestNG, SeedStack uses AssertJ. Erocci being written in Erlang uses eunit. LemonLDAP::NG, written in Perl, uses Test::More, Debian autopkgtest and the CPAN tester network.

As for the time spent on writing tests, the range of answers is very broad. 4 main groups can be distinguished: less than 5% (SystemDesigner, erocci), approximately 10% (WebLab, FusionDirectory, Hammr, SAT4J), around 20% (DocDoku, Scarbo, CLIF, SeedStack), and up to 50% (AuthzForce).

At the date of the survey, none of the surveyed projects generates test automatically.

A good practice in bug management consists in associating a test case systematically to a newly referenced bug. Half ot the projects confirm they follow this practice (DocDoku, SAT4J, Hammr, Scarbo, SeedStack, AuthzForce). The others either write a test case only above when the level of complexity is high (WebLab), or do not adopt this practice at all (nearly half ot the projects).

The tests are usually integrated in the build process during the continuous integration phase. For most projects, tests are launched during each build, which gets fired each time code gets added to the central repository. The time taken to run the full suite of unit tests can be distributed in 3 main groups: less than one minute (erocci, SeedStack, AuthzForce, CLIF, System Designer), around 2 minutes (Hammr, Scarbo, LemonLDAP::NG), 10 minutes or more: WebLab, DocDoku, SAT4J, FusionDirectory.

Another aspect relating to testing relates to the environment(s) in which the tests are executed. Several projects reproduce the diversity of the target environments using either containers (SeedStack), or using directly different operating systems running in virtual machines (WebLab, FusionDirectory), or different browser / OS combinations in the case of client-side JavaScript applications (SystemDesigner), using Karma and Saucelabs. The other surveyed projects run tests in a single environment.

Regarding the computation of test coverage, there is a significant diversity: two projects use JaCoCo (DocDoku, SAT4J) to compute this metric, WebLab uses Cobertura, erocci uses Erlang Common Test, SeedStack uses Coveralls, SystemDesigner uses Istanbul. The other projects do not compute coverage at this stage.

For most of the projects, the tests can be run directly in the IDE (IntelliJ, Eclipse) or separately in the continuous integration server. This allows the developers to have quick feedback on their code to detect broken tests due to newly changed code. Hence the tests are in majority run by the developers and by the continous integration server, except the case of CLIF which runs test only separately from the developer environment.

In terms of methodology, none of the surveyed projects uses specific framework such as Scrum or Kanban for structuring the testing phase.

In majority the tests are written after coding, except for 4 projects which adopt the Test Driven Development approach most of the time (SAT4J, erocci, AuthzForce). Along the same line, documentation is written mostly after code writing.

The maintenance of unit tests is managed in different ways: they are either part of the code itself and get hence continuously updated, or they get updated on each release. CLIF indicates that test maintenance is a problem at this stage.

For all projects, unit testing is integrated into the build step, except for FusionDirectory which does not really build software since it uses an interpreted language (PHP). This integration relies on several tools: Maven (WebLab, DocDoku, SAT4J, Scarbo, SeedStack, AuthzForce), Travis CI (Hammr), Make (erocci, LemonLDAP::NG), Grunt (SystemDesigner). Some projects complement the use of Maven with other tools: Jenkins (WebLab), Gulp (DocDoku), Travis CI (SeedStack).

Even though unit testing is important for development purpose, no project leader report that customers ask for the existence of such tests when selecting the software.

Unit tests are considered by all project leaders as absolutely needed for software quality – “I could not develop without them” says SAT4J project leader –, in particular to avoid regression and to integrate pull requests. The project leaders emphasize however that test writing implies a costly effort and that it requires a compromise between return on investment and time to market. One issue is raised regarding the difficulty to write tests that are independent of the runtime environment. The SystemDesigner project leader points out that unit tests need to comply with specific constraints such as: be easy to run, be easy to configure, come with easy to understand visualization, have the ability to re-run a test easily when it fails.

Beside unit testing, the projects use static analysis tools such as SonarQube (DocDoku, SATJ), Findbug, PMD/CD (WebLab, SAT4J, AuthzForce), Dialyzer in the case of erocci.

Several metrics are computed systematically: unit test errors and skipped tests. Tests are skipped when (i) a test method is set as to be ignored (the tag @Ignore is set) which corresponds to the case when a developer made a modification but has not updated the unit test yet, (ii) the test environment does not allow for the execution of a specific test. Two projects (SAT4J, SeedStack) monitor systematically circular dependencies, code duplication, code complexity. In addition, SAT4J monitors bugs and vulnerabilities in SonarQube.

Conclusion

The survey shows a significant diversity among the surveyed project in the way unit tests are written and run, in particular on the time spent for writing tests and for executing them. JUnit is used by all the projects written in Java, sometimes completed by additional tools such as AssertJ or Findbugs. Unit tests are always deeply integrated into the project lifecycle as a part of the continuous integration step. None of the projects generates tests automatically but all project leaders express an interest for it, which leaves room for amplifying test automatically via STAMP. All project leaders agree on one fact: unit tests are absolutely needed and are one of the cornerstones of software quality.

Chapter 10

Testing on the XWiki project

This section is a shortened version of the complete XWiki page about testing[1]. It contains the terminology and practices used in the development of their products. The general XWiki development flow [2] could be checked to better understand how testing fits in the larger picture. Here's the general methodology used:

- Code committed must have associated automated tests. There's a check in the build [3] (in the quality profile) to ensure that committed code do not reduce the total test coverage for that module. The CI runs the quality profile for each commit and fails the build if the test coverage is reduced.
- Developers run unit and integration tests on their machines daily and frequently.
- Functional tests are executed by the CI, at each commit.
- Performance tests are executed manually several times per year (usually for LTS releases and sometimes for stable releases too).
- Automated tests are currently running in a single environment (HSQLDB/Jetty/Firefox). Manual tests are executed at each release by dedicated contributors and they make sure to run them on various combinations of database, servlet containers and browsers/versions.

Unit Testing

A unit test **only tests a single class** in isolation from other classes. Since in the XWiki project the code is written using Components, this means a unit test is testing a Component in isolation from other Components.

Java Unit Testing

- These are **tests performed in isolation** using Mock Objects [4]. More specifically JUnit 4.x and Mockito [5] are being used (JMock 2.x [6] was used before and still have a lot of tests not converted to Mockito yet)
- These tests **must not interact with the environment** (Database, Container, File System, etc) and do not need any setup to execute

[1] http://dev.xwiki.org/xwiki/bin/view/Community/Testing
[2] http://dev.xwiki.org/xwiki/bin/view/Community/DevelopmentPractices#HGeneralDevelopmentFlow
[3] http://dev.xwiki.org/xwiki/bin/view/Community/Building#HAutomaticChecks
[4] http://www.mockobjects.com/
[5] https://code.google.com/p/mockito/
[6] http://jmock.org/

- These tests **must** not output anything to stdout or stderr (or it'll fail the build). Note that if the code under tests output logs, they need to be captured and possibly asserted.
- Any users' Maven module must depend on the xwiki-commons-tool-test-simple (for tests not testing Components) or xwiki-commons-tool-test-component (for tests testing Components) modules.
  - If you're testing Components, use MockitoComponentMockingRule (its javadoc explains how to use it in details)
  - Users testing non Components, should use pure JUnit and Mockito

Best practices

- Name the Test class with the name of the class under test **suffixed with Test**. For example the JUnit test class for XWikiMessageTool should be named XWikiMessageToolTest
- Name the test methods with the method to test followed by a qualifier describing the test. For example importWithHeterogeneousEncodings().

JavaScript Unit Testing

- These are **tests that do not rely on the DOM**, written as "behavioral specifications", using the Jasmine [7] test framework.
- In development mode, the test runner process can be started by running mvn jasmine:bdd in the command line. This will start a test runner at http://localhost:8234, that will run all tests in the src/test/javascript directory. Tests are written there and the results are shown directly in the browser.
- For tests that need a DOM Functional Testing [8] is employed.

Integration Testing

An integration test tests several classes together but without a running XWiki instance. For example if you have one Component which has dependencies on other Components and they are tested together this is called Integration testing.

Java Integration Testing

- These tests are written using JUnit 4.x and Mockito [9]
- Any Maven module must depend on the xwiki-commons-test module.
- The MockitoComponentMockingRule Rule is used.

Best practices

- Same as for unit testing [10]

Java Rendering Testing

There is a special framework for making it easy to write Rendering tests. [11].

[7] http://pivotal.github.io/jasmine/

[8] http://dev.xwiki.org/xwiki/bin/view/Community/Testing#HFunctionalTesting

[9] https://code.google.com/p/mockito/

[10] http://dev.xwiki.org/xwiki/bin/view/Community/Testing#HJavaUnitTesting

[11] http://rendering.xwiki.org/xwiki/bin/view/Main/Extending#HAddingTests

XAR Testing

Since XWiki 7.3M1 It's now possible to write integration tests for wiki pages on the filesystem (in XML format).

The way those tests work is that the XML file representing the wiki pages are loaded from the filesystem into XWikiDocument instances and a stubbed environment is defined so that XWikiDocument can then ben rendered in the desired syntax.

To write such a test:

- The POM file should depend on the org.xwiki.platform:xwiki-platform-test-page module
- A Java test class that extends PageTest should be written.
- Possibly some extra component registration should be added through existing annotations (*ComponentList annotations) or through the custom ComponentList annotation. This will depend on the concrete scenario of use. When extending PageTest this automatically brings some base components registration (the list defined in PageComponentList and ReferenceComponentList). Example of other existing annotations:
  - XWikiSyntax20ComponentList: XWiki Syntax 2.0-related components
  - XWikiSyntax21ComponentList: XWiki Syntax 2.1-related components
  - XHTML10ComponentList: XHTML 1.0-related components
- Then the rendering of a page is verified by setting the output syntax it should have, the query parameters and then call renderPage().

Functional Testing

A functional test requires a running XWiki instance.

GUI tests

XWiki now have 2 frameworks for running GUI tests:

- One based on Selenium2/Webdriver which is the framework to use when writing new functional UI tests.
- One based on Selenium1 which is now deprecated and shouldn't be used. XWiki committer are encouraged to port tests written for it to the Selenium2 framework. Especially when committers bring modification to the old tests they are encouraged to rewrite the tests as new Selenium2 tests.

Selenium2-based Framework

Using:

- To debug a test XE is started and then debug of JUnit tests can be done as normal JUnit tests in IDEs.
- In order to debug more easily flickering tests one can simply add the @Intermittent annotation on the test method and it'll be executed 100 times in a row. There are also annotations to control the number of executions. This is achieved thanks to the Tempus Fugit framework [12].
  - To run specific tests regex are employed by passing a pattern property parameter. Details on the usage of the pattern property and examples can be found in the documentation [13].

[12] http://tempusfugitlibrary.org/documentation/junit/intermittent/

[13] https://github.com/xwiki/xwiki-platform/blob/master/xwiki-platform-core/xwiki-platform-test/xwiki-platform-test-integration/src/main/java/org/xwiki/test/integration/XWikiExecutorTestMethodFilter.java#L25

- Users can also run the tests on their own running instance.
- By default the Firefox browser will be used but it can changed with another browser by just passing the browser parameter as in:
  - Firefox (default): -Dbrowser=*firefox
  - Internet Explorer: -Dbrowser=*iexplore
  - Chrome: -Dbrowser=*chrome
  - PhantomJS: -Dbrowser=*phantomjs
- Compatibility problems may arise between the versions of the browser and Selenium.

Best practices

- Tests are located in xwiki-platform inside the module that they are testing. In the past functional tests were included in xwiki-enterprise-test-ui but have been moved to the xwiki-platform module.
- Tests should be written using the Page Objects Pattern [14] [15]
- Since functional tests take a long time to execute (XWiki to start, Browser to start, navigation, etc) it's important to write tests that execute as fast as possible. Here are some tips considered:
  - Write scenarios (i.e. write only a few test methods, even only one) instead of a lot of individual tests as it is usually done for unit tests.
  - getUtil() is used to perform quick actions that are not part of what is being tested (i.e that are part of the test fixture).
  - Maven modules for test may depend on a specific profile (as a best practice, should use integration-tests). Doing this, it will be built only when specifically asked with -Pintegration-tests .
- There is a need to create, update or delete a page during tests, a space name specific to test scenario should be specified.
- Never, ever, wait on a timer! Instead wait on elements.
- Tests must not depend on one another. In other words, it should be possible to execute tests in any order and running only one test class should work fine.
- Tests that need to change existing configuration (e.g. change the default language, set specific rights, etc) must put back the configuration as it was.
- Tests are allowed to create new documents and don't need to remove them at the end of the test.

The Office Importer tests

XWiki 7.3, have introduced in xwiki-platform some functional tests for the Office Importer Application [16]. To enable them, you need to enable the profile office-tests. An OpenOffice (or LibreOffice) server is needed on the target system.

Old Selenium1-based Framework

- Selenium RC is used to perform functional tests for GUI. Some JUnit extension has been created to easily write Selenium tests in Java.
  - Existing tests can be found in the corresponding page [17]xwiki-enteprise/xwiki-enterprise-test/xwiki-enterprise-test-selenium
- There are options and functionalities available to run tests locally, run an specific test, enable output debugging from the Selenium server, and debug functional tests in IDEs.

[14] https://code.google.com/p/selenium/wiki/PageObjects
[15] https://code.google.com/p/selenium/wiki/PageObjects
[16] http://extensions.xwiki.org/xwiki/bin/view/Extension/Office%20Importer%20Application
[17] https://github.com/xwiki/xwiki-enterprise/tree/master/xwiki-enterprise-test/xwiki-enterprise-test-selenium

Browser version

Functional tests are run only on the Firefox browser.
Any other browser used for functional tests needs to match the selenium version, otherwise unpredictable results may occur.

- The current Selenium version used is 2.44.0 [18]
- The Firefox version used in continuous integration agents is 32.0.1. This version is needed to run tests in the exact configuration as XWiki's Continuos Integration Server[19]

XHTML, CSS & WCAG Validations

- JUnit is used to validate that all XE pages produce valid XHTML.
  - Existing tests can be found in enterprise/distribution-test/misc-tests/
- WCAG Testing Strategy [20]
- CSS validation must be verified manually using the W3C validator [21]. An specific coding style is advised. [22]

Performance Testing

- These are memory leakage tests, load tests and speed of execution tests.
- They are performed manually and in an ad hoc fashion for now. They are executed for some stable versions and for all super stable versions.
- See Methodology and reports.

See the Profiling topic for details on how to use a profiler for detecting performance issues.

Manual testing

Besides automated testing, ensuring software quality requires that features be tested by actual users, that is to perform manual tests. In order to manage manual testing, a test plan is required. The overall strategy is available for developers and contributors [23]. Manual tests target mostly cross-browser and cross-platform scenarios. All test cases should be described and reported using the appropriated tools [24]. The spirit XWiki would like to have from the XWiki Manual Testing Team can be found in: The Black Team[25].

Tools Reference

The following tools are used in automated tests:

- JUnit: test framework

[18] valid for 18.March.2015. Actual version used can be verified in https://github.com/xwiki/xwiki-commons/blob/master/pom.xml
[19] http://ci.xwiki.org/
[20] http://dev.xwiki.org/xwiki/bin/view/Community/WCAGTesting
[21] http://jigsaw.w3.org/css-validator/
[22] http://dev.xwiki.org/xwiki/bin/view/Community/XhtmlCssCodeStyle
[23] http://dev.xwiki.org/xwiki/bin/view/Community/ManualTestStrategy
[24] http://test.xwiki.org/xwiki/bin/view/Main/
[25] http://www.t3.org/tangledwebs/07/tw0706.html

- Mockito, JMock: mocking environment of unit test to isolate it
- Hamcrest: extra assertions beyond what JUnit provides
- GreenMail [26]: for testing email sending
- WireMock [27]: for simulating HTTP connections to remote servers
- JMeter footnotehttp://jmeter.apache.org/: for performance tests
- Dumbbench [28]: for manual performance tests

Test Coverage

A SonarQube instance [29] showing Test coverage for both unit tests and integration tests is available. However it doesn't aggregate coverage data across top level modules (commons, rendering, platform, enterprise, etc).

XWiki supports both Jacoco[30] and Clover[31] to generate test coverage reports.

[26] http://www.icegreen.com/greenmail/

[27] http://wiremock.org/

[28] https://github.com/tsee/dumbbench

[29] http://sonar.xwiki.org/

[30] http://www.eclemma.org/jacoco/

[31] http://www.atlassian.com/software/clover/overview

Chapter 11

Conclusion

We presented a survey of works related to test amplification and practices from industrial partners. The survey is the first that draws a comprehensive picture of the different engineering goals proposed in the literature for test amplification, which goes far beyond maximizing coverage. It also gives an overview of the different techniques used, which span a wide spectrum, from symbolic execution to random search to execution modification. We believe that this survey 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.

This document also sets some of the basis for collaboration and discussion between the partners. It provides a good insight of the main technologies in use and serves as a guide for understanding the gap between theory and practice in the context of software testing.

Bibliography

[1] 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.

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

[3] B. Baudry, F. Fleurey, J.-M. J'ez'equel, and L. Yves. From genetic to bacteriological algorithms for mutation-based testing. *Software, Testing, Verification & Reliability journal (STVR)*, 15(2):73–96, June 2005.

[4] 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, New York, NY, USA, 2006. ACM.

[5] R. Bloem, R. Koenighofer, F. RÃűck, and M. Tautschnig. Automating test-suite augmentation. In *2014 14th International Conference on Quality Software*, pages 67–72, Oct 2014.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] M. R. Marri, S. Thummalapenta, T. Xie, N. Tillmann, and J. de Halleux. Retrofitting unit tests for parameterized unit testing. Technical report, Technical Report TR-2010-9, North Carolina State University Department of Computer Science, Raleigh, NC, 2010.

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

[19] 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.

[20] M. Mirzaaghaei, F. Pastore, and M. Pezze. Supporting Test Suite Evolution through Test Case Adaptation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 231–240, 2012.

[21] M. Mirzaaghaei, F. Pastore, and M. Pezzè. Automatic test case evolution. *Software Testing, Verification and Reliability*, 2014.

[22] T. Mouelhi, Y. L. Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing. In *2009 International Conference on Software Testing Verification and Validation*, pages 171–180, April 2009.

[23] C. Pacheco and M. D. Ernst. *Eclat: Automatic Generation and Classification of Test Inputs*, pages 504–527. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[24] 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.

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

[26] 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.

[27] 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.

[28] J. Röβler, 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.

[29] 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.

[30] 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.

[31] B. H. Smith and L. Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3):341–369, 2009.

[32] B. H. Smith and L. Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, 2009.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

[43] J. Xuan, M. Martinez, F. DeMarco, M. ClÃl’ment, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] Z. Yu, C. Bai, and K.-Y. Cai. Mutation-oriented Test Data Augmentation for GUI Software Fault Localization. *Inf. Softw. Technol.*, 55(12):2076–2098, Dec. 2013.

[50] 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.

[51] 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.