Master research Internship

Internship report

Adapting Amplified Unit Tests for Human Comprehension

Domain: Software Engineering — Artificial Intelligence

Author:
Simon Bihel

Supervisor:
Benoit Baudry

KTH Royal Institute of Technology

Abstract:

TODO

Contents

Introduction **1**

1 Background **1**
1.1 Software Testing . . . 1
1.1.1 Test Activities . . . 1
1.1.2 Levels of Testing . . . 2
1.1.3 Unit Testing . . . 2
1.2 Elementary Metrics . . . 2
1.3 Mutation Testing . . . 3
1.4 The Need for Easy-to-use Tools . . . 4
1.5 Cognitive Support Tool Development . . . 4

2 Test Suite Amplification **4**
2.1 Genetic Improvement . . . 4
2.2 Test Amplification . . . 4
2.3 DSpot . . . 5

3 Problem Statement **5**
3.1 The Need for Unit Test Cases Documentation . . . 5
3.2 The Generated Random Noise . . . 5

4 Related Works **5**
4.1 Automatic Test Case Documentation . . . 6
4.2 Cognitive Support for Unit Testing Review . . . 6

5 Contribution **6**
5.1 Identifying Amplifications . . . 6
5.2 Minimisation . . . 6
5.3 Replace or Keep . . . 6
5.4 Focus . . . 6
5.5 Slicing . . . 6
5.6 Natural Language Description . . . 6
5.7 Ranking . . . 7

6 Evaluation **7**
6.1 Threat to Validity . . . 7

Conclusion **7**

Acknowledgments **7**

References **10**

Introduction

TODO

1 Background

In this section, we present the landscape this thesis fits into. We define testing (Section 1.1) as well as go over its use in the industry. In particular, to understand the needs of practitioners, we present how they assess the quality of their code (Sections 1.2 and 1.3) and what it takes for a new tool to be incorporated in their workflow (Sections 1.4 and 4.2).

1.1 Software Testing

In this section we give a definition for the test activities and their actors, the different abstraction levels of tests, and our precise object of study, unit tests.

TODO: Why are we testing software and how do we do it

TODO: Say that the formal definitions aren't totally necessary

1.1.1 Test Activities

TODO TODO: the text is too close to the oracle survey

Testing is about verifying that a system, for a certain scenario, follows a certain behaviour that was previously defined. The *System-Under-Test* (SUT), in our domain a software system, has a set of components C. A scenario is sequence of stimuli that target a subset of components, and trigger both responses from the SUT and changes in the state of its components. From [1], we have the following definition for test activities:

TODO: can you add a subjective judgement here? like "here is a well written definition"

Definition 1.1 (Test Activities)**.** For the SUT p, S is the set of stimuli that trigger or constrain p's computation and R is the set of observable responses to a stimulus of p. S and R are disjoint. Test activities form the set $A = S$.

The use of the terms "stimulus" and "observations" fit various test scenarios, functional and non-functional. Figure 1 depicts the fact that a stimulus can be either an explicit input from the tester, or an environmental factor that can affect the SUT. TODO

Definition 1.2 (Test Case)**.** TODO

Definition 1.3 (Test Suite)**.** A test suite is a collection of tests cases.

Definition 1.4 (Test Oracle)**.** A test oracle $D : T_A \mapsto \mathbb{B}$ is a partial function from a test activity sequence to true or false.

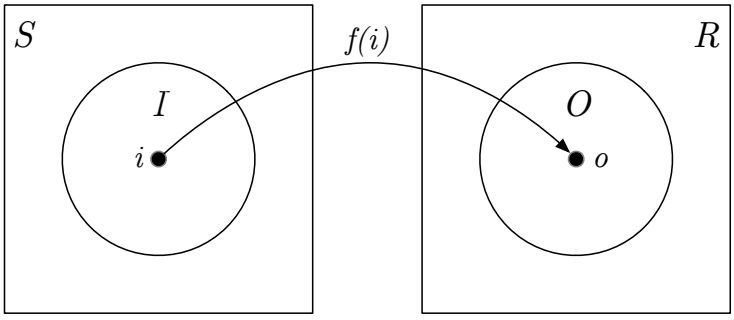

Figure 1: Stimuli and observations: S is anything that can change the observable behaviour of the SUT f; R is anything that can be observed about the system's behaviour; I includes f's explicit inputs; O is its explicit outputs; everything not in $S \cup R$ neither affects nor is affected by f.

Another concept that we will use when talking about software evolution is regression testing [2]:

Definition 1.5 (Regression Testing)**.** Regression testing is performed between two different versions of the SUT in order to provide confidence that newly introduced features do not interfere with the existing features.

TODO: give some clues to each concept to tell why it is a field of research?

1.1.2 Levels of Testing

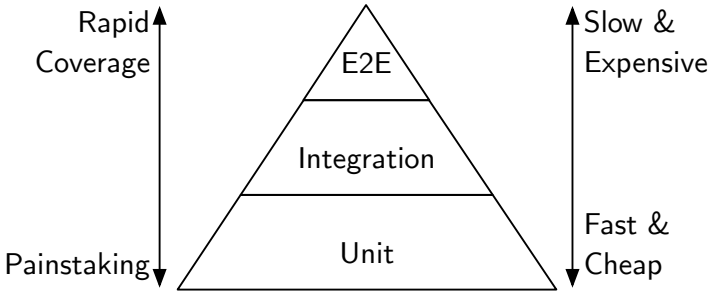

Figure 2: A view of the test pyramid.

In the case of software testing, the SUT can be different elements of a program, i.e. it can belong to different levels of abstraction. One can decide to test a whole program, by manually giving inputs and verifying that the output is correct. Or one could test individual functions, having a more control over the behaviour of each elements.

These levels of test abstraction can be visualised with a test pyramid, as shown in Figure 2. Tests are generally separated into three distinct categories. The top category is for the End-to-End tests, which means testing the program as a whole. Whilst it allows for directly evaluating the well behaviour of the program, each test is costly in time and resources because the whole program has to be run. Not only will each component be run, whether or not it requires testing, but time will also be spent on call to libraries that are already tested. At the other end of the test spectrum are the unit tests. The goal is to test each and every component of the program (e.g. every function), as an individual and isolated from the rest of the program. Running a single component is faster than running the whole program, but require much more work from the person writing tests, as every component needs its set of tests. In between the two testing practices are the integration tests, which aim at ensuring the well collaboration of components.

TODO: metaphor about rotating gears?

TODO: how tests are automated

1.1.3 Unit Testing

A unit test is test for a precise component of the SUT, but defining the level of granularity is not trivial, even for experienced programmers [3]. **TODO**

An example of unit test is given in Figure 3.

TODO: why we sometimes speak about test methods

1.2 Elementary Metrics

Metrics have been developed to help developers automatically assess the quality of their code. Each metric measures one specific characteristic, such as the need for refactoring, the size of the program, or — and we are particularly interested in this one — the likelihood of bugs.

```
1  testIterationOrder() {
2    TreeList tl = new TreeList(10);
3    for (int i = 0; i < size; i++) {
4      tl.add(i);
5    }
6    int i = 0;
7    ListIterator it = tl.listIterator();
8    while (it.hasNext()) {
9      Integer val = it.next();
10     assertEquals(i++, val.intValue());
11   }
12 }
```

Figure 3: Example of a unit test: it consists of test inputs that manipulate the SUT; and assertions (line number 10).

TODO: give examples for other metrics?

The most basic metrics to measure how thoroughly tested a system is, are coverage based metrics. For example, during the test suite execution, one can keep track of all statements that were executed. At the end, you have a percentage of executed statements for the total number of statements. Instead of statements, one can also keep track of control flow branches explored.

It is generally acknowledged that a system with high coverage means that the system is less likely to have bugs. But it is not foolproof.

TODO: What are their limits

- doesn't mean corner cases are avoided
- can't have 100% branch coverage so it's confusing

1.3 Mutation Testing

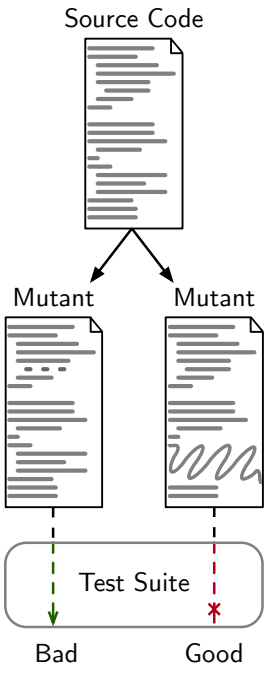

Figure 4: Mutation testing process

Mutation testing is a method used to assess the quality of a test suite, giving it a grade called *mutation score*. The idea is to insert bugs in the software and see if these faulty versions still pass the tests or not. We call the derived versions *mutants*, hence the name mutation testing. When a mutant goes through the test suite without making a single test fail, it is considered *live*, and the opposite case it is considered *dead*. The global process is depicted in Figure 5. The main appeal is that, a test suite with a good mutation score, in addition to ensure the well behaviour of the SUT in useful scenarios, it can detect bad behaviours — which is particularly useful for regression testing. **TODO: rework that sentence** There is a correlation between mutation score and coverage score as test suites with high mutation score tend to have good coverage [4]. It has been found that correlations between mutation scores and real fault detection are weak [5].

[fundational papers]

Mutants are obtained by applying *mutators* on the original version. These operators can be varied: change a condition (e.g. substituting $\leq$ with $>$), delete the body of a method [descartes], etc.

Mutation testing suffers from drawbacks that have limited its use outside of the research world even though it has been an active research domain for many years [6]. The first obvious flaw is that it is slow. For compiled languages, all mutants have to be compiled, and that process often takes a lot of time for large programs. Then the whole test suite has to be executed for each mutant, which again is a long process. However, given certain trade-offs in terms of quality of the mutation analysis process, the computational complexity can be dodged [7]. Optimized techniques for regression testing also exist [2], including minimisation, selection and prioritisation. The mutant generation is also full of traps. The major pitfall that cannot be avoided is the equivalent mutant problem. Some mutants, although they have a different program than the original, have the same semantics. Which means that they will always pass the test suite and will not bring any insight on the SUT.

Another difficulty for mutation testing to take on in the industry is the lack of understanding by practitioners. TODO: rework that sentence The process of mutation and elimination can be confusing. But also, when a mutant is live, it is not always clear what actions should be taken in order to kill it. TODO: maybe add more

PIT[1] [8] is a mutation testing tool for Java.

TODO: what is great about it

TODO: maybe quick comparison with other tools

TODO: it is deterministic

1.4 The Need for Easy-to-use Tools

TODO: easy to understand TODO: useful for surveys [9]

1.5 Cognitive Support Tool Development

TODO [10] [11]

2 Test Suite Amplification

In this Section, we provide more context for this thesis' contribution. TODO

2.1 Genetic Improvement

TODO [12] [fundational papers]

[the surprising creativity of digital evolution?]

2.2 Test Amplification

TODO [13]

[1] http://pitest.org/

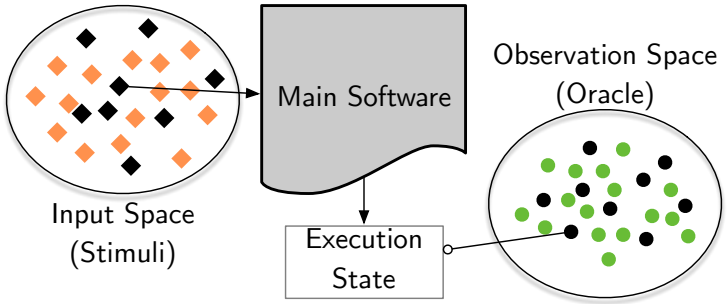

Figure 5: On the left, the testing input space is composed by specified input points (orange diamonds) and unspecified input points (black diamonds). On the right, the observation space over a given program state depicts the assertions of tests. The green circles are values that are already asserted in the existing test suite, the newly added assertions are shown as black circles.

2.3 DSpot

TODO [2] [dspot papers] [14, 15, 16] [17]

3 Problem Statement

This thesis aim at helping developers understand amplified tests. There are two sides for this problem: supporting the developer understand the test (Section 3.1) and cleaning the tests from the noise injected during the amplification process (Section 3.2).

3.1 The Need for Unit Test Cases Documentation

TODO Several works have highlighted the need for test documentation [18, 19, 20, 21, 22, 23]. [ ]

TODO: tests are complex

TODO: lack of why information

TODO: the code the test interacts with; what the code does; what is expected

TODO: programmers like short textual description

TODO: explain what li2016automatically has done

TODO: talk about the field of software maintenance

3.2 The Generated Random Noise

TODO

4 Related Works

TODO

[2] https://github.com/STAMP-project/dspot

4.1 Automatic Test Case Documentation

TODO

TODO: paraphrasing the code; lack of why information

TODO: stereotypes are for general purpose code

[24, 25, 21, 26, 27, 28]

TODO: there are also works on documenting code changes (i.e. commits) [29, 30, 31, 32, 33, 34, 35]

4.2 Cognitive Support for Unit Testing Review

TODO

5 Contribution

TODO

5.1 Identifying Amplifications

TODO

5.2 Minimisation

TODO TODO: put slicing before?

TODO: removing useless assertions

TODO: cannot use general purpose techniques[36, 37] because we want the original part intact

5.3 Replace or Keep

TODO

5.4 Focus

TODO

5.5 Slicing

TODO [38][3]

5.6 Natural Language Description

TODO

TODO: Focus on mutation testing

TODO: avoid talking about mutants TODO: How to describe a mutant

[39, 40, 41]

[3] http://wala.sourceforge.net/

List of possible information:

- control flow branches
- name of mutant
- code under test [42]

TODO: on the usefulness of nlg

5.7 Ranking

TODO

6 Evaluation

TODO

6.1 Threat to Validity

TODO

TODO: DSpot is not yet established and recognized in the community. It's is difficult to have

Conclusion

TODO

Acknowledgments

Thanks to Benoit Baudry and Martin Monperrus for their guidance. Thanks to Benjamin Danglot for his collaboration and all his work on DSpot. Thanks to Zimin Chen, Nicolas Harrand, He Ye and Long Zhang for making daily life enjoyable. This internship was supported by the Fondation Rennes 1 and its patrons. Many thanks to KTH for hosting me.

References

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[3] P. Runeson, "A survey of unit testing practices," *IEEE software*, vol. 23, no. 4, pp. 22–29, 2006.

[4] B. Assylbekov, E. Gaspar, N. Uddin, and P. Egan, "Investigating the correlation between mutation score and coverage score," in *Computer Modelling and Simulation (UKSim), 2013 UKSim 15th International Conference on*, pp. 347–352, IEEE, 2013.

[5] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *40th International Conference on Software Engineering, May 27-3 June 2018, Gothenburg, Sweden*, 2018.

[6] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[7] J. Možucha and B. Rossi, "Is mutation testing ready to be adopted industry-wide?," in *International Conference on Product-Focused Software Process Improvement*, pp. 217–232, Springer, 2016.

[8] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 449–452, ACM, 2016.

[9] M. Delahaye and L. Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *Software: Practice and Experience*, vol. 45, no. 7, pp. 875–891, 2015.

[10] S. Oviatt, "Human-centered design meets cognitive load theory: designing interfaces that help people think," in *Proceedings of the 14th ACM international conference on Multimedia*, pp. 871–880, ACM, 2006.

[11] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: a critical review and guidelines," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 120–131, IEEE, 2016.

[12] J. Petke, S. Haraldsson, M. Harman, D. White, J. Woodward, *et al.*, "Genetic improvement of software: a comprehensive survey," *IEEE Transactions on Evolutionary Computation*, 2017.

[13] B. Danglot, O. Vera-Perez, Z. Yu, M. Monperrus, and B. Baudry, "The emerging field of test amplification: A survey," *arXiv preprint arXiv:1705.10692*, 2017.

[14] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus, "Automatic software diversity in the light of test suites," *arXiv preprint arXiv:1509.00144*, 2015.

[15] B. Baudry, S. Allier, and M. Monperrus, "Tailored source code transformations to synthesize computationally diverse program variants," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 149–159, ACM, 2014.

[16] B. Baudry, S. Allier, M. Rodriguez-Cancio, and M. Monperrus, "Dspot: Test amplification for automatic assessment of computational diversity," *arXiv preprint arXiv:1503.05807*, 2015.

[17] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.

[18] M. P. Prado, E. Verbeek, M.-A. Storey, and A. M. Vincenzi, "Wap: Cognitive aspects in unit testing: The hunting game and the hunter's perspective," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pp. 387–392, IEEE, 2015.

[19] M. P. Prado and A. M. Vincenzi, "Advances in the characterization of cognitive support for unit testing: The bug-hunting game and the visualization arsenal," in *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*, pp. 213–220, IEEE, 2016.

[20] M. P. Prado and A. M. R. Vincenzi, "Towards cognitive support for unit testing: a qualitative study with practitioners," *Journal of Systems and Software*, 2018.

[21] B. Li, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and N. A. Kraft, "Automatically documenting unit test cases," in *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pp. 341–352, IEEE, 2016.

[22] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pp. 201–211, IEEE, 2014.

[23] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 547–558, IEEE, 2016.

[24] G. Neubig, "Survey of methods to generate natural language from source code," 2016.

[25] N. Nazar, Y. Hu, and H. Jiang, "Summarizing software artifacts: A literature review," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 883–909, 2016.

[26] B. Li, *Automatically Documenting Software Artifacts*. PhD thesis, College of William & Mary, 2018.

[27] M. Kamimura and G. C. Murphy, "Towards generating human-oriented summaries of unit test cases," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pp. 215–218, IEEE, 2013.

[28] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pp. 61–70, IEEE, 2015.

[29] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pp. 275–284, IEEE, 2014.

[30] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2, pp. 709–712, IEEE, 2015.

[31] N. Dragan, M. L. Collard, M. Hammad, and J. I. Maletic, "Using stereotypes to help characterize commits," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 520–523, IEEE, 2011.

[32] S. Jiang and C. McMillan, "Towards automatic generation of short summaries of commits," in *Proceedings of the 25th International Conference on Program Comprehension*, pp. 320–323, IEEE Press, 2017.

[33] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 135–146, IEEE Press, 2017.

[34] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, "On automatic summarization of what and why information in source code changes," in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 1, pp. 103–112, IEEE, 2016.

[35] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 33–42, ACM, 2010.

[36] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 417–420, ACM, 2007.

[37] A. Zeller, "Yesterday, my program worked. today, it does not. why?," in *ACM SIGSOFT Software engineering notes*, vol. 24, pp. 253–267, Springer-Verlag, 1999.

[38] J. Dolby, S. J. Fink, and M. Sridharan, "Tj watson libraries for analysis (wala)," *URL http://wala. sf. net*, 2015.

[39] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 182–191, IEEE, 2008.

[40] L. P. Hattori and M. Lanza, "On the nature of commits," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. III–63, IEEE Press, 2008.

[41] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and software*, vol. 7, no. 4, pp. 325–339, 1987.

[42] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Scotch: Test-to-code traceability using slicing and conceptual coupling," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pp. 63–72, IEEE, 2011.