Software Regression as Change of Input Partitioning

Marcel Böhme*
School of Computing
National University of Singapore
marcel.boehme@nus.edu.sg

***Abstract*—It has been known for more than 20 years. If the subdomains are not homogeneous, partition testing strategies, such as branch or statement testing, do neither perform significantly better than random input generation nor do they inspire confidence when a test suite succeeds. Yet, measuring the adequacy of test suites in terms of code coverage is still considered a common practice. The main target of our research is to develop strategies for the automatic evolution of a test suite that does inspire confidence. When the program is changed, test cases shall be augmented that witness changed output for the same input (test suite augmentation). If two test cases witness the same partition, one is to be discarded (test suite reduction).**

***Keywords*-Partition Testing; Software Evolution; Reliability; Automated Test Generation**

I. Research Problem

A. Introduction

Software regresses when existing functionality that used to work does not anymore. The aim of regression testing is to create and maintain a test suite that stresses much of the program's behavior, so that when the program is changed at least one test case should fail upon execution on the changed program when the behavior regresses. The adequacy of test suites can be measured in terms of code coverage. For instance, a test suite achieves hundred percent branch coverage if every branch in the program is exercised by at least one test in the test suite. Branch testing devides the input space into (non-disjoint) subdomains [1], with every subdomain consisting of all test cases that exercise a particular branch. There is one subdomain per branch. A test suite is branch-coverage adequate if it contains at least one test case of each subdomain. Testing strategies that devide the input space into classes whose points are somehow "the same" are summarized under partition testing [2].

Code coverage is an insufficient measure of adequacy to determine whether a test suite exposes software regression; so is mutation-based or specification-based, functional testing. The respective subdomains are *not homogeneous w.r.t. failure*. For instance, it is not true that if a test case exercises a certain branch and exposes an error, then every test case exercising the same branch exposes an error. Hamlet et al. [2] argue that partition testing shall not inspire confidence in program correctness, even if all tests of an adequate test suite run successful. Duran and Ntafos [3] experimentally show that partition testing does not perform significantly better in terms of revealing an error than random input (that does not follow a partitioning scheme). Weyuker et al. [1] theoretically analyzed the results and observe that the effectiveness of partition testing varies substantially and conclude that "partition testing is most successful when the sub-domain definitions are fault-based", that is, *homogeneous w.r.t. failure* - if one test fails, every test in the same partition fails. Interestingly, if the input spaces of original and changed program are devided into (disjoint) homogeneous subdomains, then we could expose software regression as well as progression at the intersection of the subdomains of original and changed program. Software progression is observed if input that used to fail passes in the changed program.

In summary, if the subdomains are not homogeneous, partition testing does neither perform significantly better than random input generation nor does it inspire confidence when a test suite succeeds. Yet, measuring the adequacy of test suites in terms of code coverage is still considered a common practice. *The main target of our research is to develop strategies for the automatic evolution of a test suite that does inspire confidence*. When the program is changed, test cases shall be augmented that witness changed output for the same input (test suite augmentation). If two test cases witness the same partition, one is to be discarded (test suite reduction).

Only recently, we have begun to understand the definition and automatic exploration of homogeneous subdomains. Godefroid et al. [4] introduce dynamic symbolic execution and show that if a concrete test case exercises a certain path, then every test case satisfying the same path condition exercises that path. Qi et al. [5] introduce path exploration based on symbolic output and show that if a concrete test case computes some symbolic output, then every test case satisfying the same relevant slice condition computes the same symbolic output. We can say, that such partitions are homogeneous w.r.t. program behaviour. Using dynamic program dependencies, symbolic execution, and constraint solving, test cases are automatically generated that witness "adjacent" partitions.

We want to explain software pro- and regression as change of partitioning. A set of inputs that belongs to one partition and uniformly computes some symbolic output in program P may belong to different partitions and compute

* Research Profile: http://comp.nus.edu.sg/~mboehme

different symbolic output in changed program P'. Using the program dependencies, e.g., the relevant slice, we can determine those partitions affected by the changes. Using logic operators and constraint solving, we can find input that is at the intersection of the partitions of P and P' that compute different symbolic output. Thereby, test cases witnessing the old partitions can be reused as witnesses of some of the new partitions. The remainder of the new partitioning shall further be explored. As of now, it is not possible to predict the new partitioning. This may be one of the reasons why non-evolving test suites are inadequate w.r.t. exposing software regression. For example, if after a change, one partition is "split" into two partitions, one of which computes the "wrong" output. The test case witnessing the old partition may or may not expose this regression.

Whether the program functionality pro- or regresses can be observed using input for which the output is computed differently after the program has changed. Dually, input that computes the same symbolic output before and after a program changed, guarantees preserved functionality. Changed output is derived by overlapping old and new partitions. The behavior has changed for intersecting input partitions for which an input computes different symbolic output.

B. Expected Contributions

The following overviews my current idea about the research agenda towards the dissertation and the expected contributions to relevant fields in Testing and Debugging.

1) **Test Suite Augmentation:** Assuming an existing test suite determines that a program P behaves correctly. When the program is changed to P', add those test cases to the test suite that expose a semantic difference when executed on P and P'. In other words, for P and P' the computed output should be different at the intersection of input partitions which are homogeneous w.r.t. *program behavior*.
2) **Fault-based Test Suite Evolution:** Continously and incrementally generate test cases as witnesses of unexplored, input partitions that are homogeneous w.r.t. *program failure*. When the program is changed, automatically remove test cases witnessing affected input partitions and add those that witness new partitions. Software regression can be derived from the intersecting subdomains of new and old partitioning.
3) **Debugging** [*Optional*]**:** Input partitions that are homogeneous w.r.t. failure imply extensive knowledge about when the program fails and when it does not. Statements executed by many failing test cases are positive evidence of a fault location (cp. [6]), while statements executed by both, passing and failing test cases are negative evidence (cp. [7], [8]). Faults shall be localized automatically *while* the program is automatically tested in a fault-based manner.
4) **Predicting Regression** [*Optional*]**:** Given a "correct" program and a set of program changes. Is it possible to predict, whether the changes introduce any bugs? In other words, given a "correct" program, a set of changes, and the "old" input partitions affected by the change. Is it possible to predict the new partitioning of the input space affected by the changes?

II. Related Work

This section gives a compact and incomplete overview on works related to test suite augmentation and evolution. In Test Suite Augmentation (TSA), we want to generate test cases that witness the impact of a change onto the output. These test cases shall execute at least one changed statement, at least one output statement and compute the output differently for the old and new version of the program [9].

Change Impact: It is possible to statically determine statements onto which a change has *definitely no impact*[1]. Otherwise, via static analysis it can only be determined that the change *may or may not have an impact* on a statement for a particular execution[2]. For the same reason Chianti, the change impact tool of Ren et al. [11], can statically only determine which test cases do *not* execute a change (cp. test selection) and which changes are *not* executed by a test case. **Change Execution:** A change does only impact subsequent statements (including the output) for test cases that actually exercise the changed statement [12]. To determine whether or not a statement can actually be exercised at all is called infeasible path problem and generally undecidable [13]. However, there exist several approaches that generate input incrementally until a path is found that executes a given statement [14], [15]. Taneja et al. discuss eXpress as approach to generate test cases for every program path that exercises a changed statement in a changed program [16]. **Change Propagation:** To witness the impact of the change on the output, the generated test case shall Execute the change, Infect the program state, and Propagate the effects of the change to the output - the PIE requirements [17]. Santelices et al. [18] discuss the propagation of the effects of a change along a dependence chain to a given maximal distance. Qi et al. [14] examine an efficient approach of executing and propagating a single change to the output. Harmann et al. [19] show the execution and propagation of multiple changes to the output and acknowledge that incrementally executing every combination of the set of program changes would be prohibitively expensive. **Semantic Approach to TSA:** Person et al. introduce Differential Symbolic Execution (DSE) to derive differential method summaries [20]. A method summary is an attempt towards the complete account of behavioral difference. However, information that is relevant to actually generate input that

[1]cp. syntactic dependence [10].

[2]cp. syntactic as approximation of semantic dependence [10]

witnesses behavioral difference is removed by the abstracting uninterpreted functions. A detailed discussion of related work pertaining TSA can be found in [9].

Test Suite Evolution (TSE) integrates the automatic reduction and augmentation of test cases to re-establish the adequacy of a minimal test suite when a program evolves. A test suite T is minimal if removing a test case from T would reduce its adequacy. While there are many works on test suite reduction and augmentation, we are not aware of one that integrates both as automatic test suite evolution approach. Mirzaaghaei [21] presents an idea on repairing a test suite according to patterns they have observed when developers repair test suites when a program is changed. A repair pattern example is: *Introduction of Overloaded Method*. While this work is called Automatic Test Suite Evolution, our intention is a more general, paradigm-independent, approach towards test suite evolution.

For the lack of space, we only want to mention mutation adequacy as instance of predicting software regression. A test suite is mutation adequate if, when the program is changed, at least one test case executes these changes (i.e., weakly killing a mutant) and propagates their effects to the output (i.e., strongly killing a mutant; cp. [19]). As such, a "good" (mutation-adequate, non-evolving) test suite should anticipate any program change to be better prepared for detecting software regression. The intention of our future work is to predict the new partitioning of the input domain, given a change to a statement. This enables the prediction of software pro- or regression for any program change.

III. Our Approach

A. Partitioning in Terms of Computation and Reachability

At the core of our approach is the partitioning of the input space into subdomains that are homogeneous w.r.t. a certain property like *reachability* or *computation* of a certain statement like the output or a change.

Qi et al. [5] started foundational, theoretical work on partitioning the input space in terms of the *computation* of the output. Every input in the same partition computes the same symbolic output. Statement instances that contribute in computing the symbolic output value for some input are summarized as *relevant slice*[3] of the output statement instance. The condition computed upon program input that yields the same relevant slice is called *relevant slice condition*[4]. The authors also provide an algorithm, that explores these subdomains to find all symbolic values the output can have. Qi et al. prove *homogenity* of the subdomains when the input is partitioned based on relevant slice conditions and the *completeness* of the provided algorithm to explore all of these subdomains.

[3]Definition of relevant slice see [22].

[4]Definition of relevant slice condition see [5].

In [9], we discuss the partitioning of the input space in terms of the *reachability* of a given statement. Every input that does (not) exercise a given statement "for the same reason" is in the same partition. The condition to be satisfied for input in the same partition is called *reachability condition*[5]. We adapt the exploration algorithm of Qi et al. [5] and prove the *homogenity* of the subdomains when the input is partitioned based on the reachability condition.

It is possible to sub-partition subdomains based on relevance. In TSA, only paths that exercise at least one changed statement and one output can possibly propagate the impact of a change to the output. Paths that do not exercise a changed statement are less relevant. So, the input space is first partitioned in terms of the reachability of a change. If an input does not exercise a change, the algorithm explores partitions adjacent to witnessed reachability partition of the change. If the input does exercise a changed statement, the algorithm further explores reachability and ultimately computation subpartitions of the output within this reachability partition of the exercise change. The partitioning is based on the *change condition*[6] which basically is the conjunction of the reachability conditions of every change and output and the relevant slice conditions of the exercise output statement instances if at least one change is exercised. The change condition is proven to yield homogeneous subdomains. If an input executes a set of changed statement instances C_i, a set of output statement instances O_i and computes symbolic values V for the variables used in O_i, then every input in the same change partition exercises C_i and O_i and computes V for O_i.

B. Software Regression as Change of Partitioning

Software regression is found in the part of the input space that is affected by a program change. If for the same input the output is the same, we cannot observe any behavioral difference. It follows, in order to observe a semantic program change, for overlapping input partitions the symbolic output states must be different.

Our approach to test suite augmentation [9] partitions the input into homogeneous subdomains based on the change condition yielding change partitions. We determine for every change partition the output state computed after exercising at least one changed statement for both, the original and changed version of the program. Then, the semantic difference is determined so that for overlapping change partitions, the computed output state must be different. The generated test cases are witnesses of the behavioral difference of both programs. *Semantic change* is defined formally based on the notion of change partitions overlapping for original and changed program. Both, the change condition and the program output are quantifier-free first-order formulae on

[5]Definition of reachability condition see [9].

[6]Definition of change condition see [9].

the input variables. For every change partition of P and P', it is solved the conjunction of both change conditions and that both symbolic output values are to be different[7]. Furthermore, it is shown that the sole evaluation of the changed program while disregarding the original program is insufficient and propose the number of exposed semantic changes as measure of adequacy for augmented test suites.

Our research on automatic test suite evolution is ongoing. As intermediate result, we can report that is possible to concisely derive the affected input partitions, when the program is changed. Gyimóthy et al. [22] discuss the relevant slice as the set of statements that are involved in computing the symbolic value of a statement instance. Thus, if a statement is changed which is in the relevant slice of the output, the input partition is affected that exercises this relevant slice. It is possible to start with random test generation to achieve an initial saturation of witnessed change partitions in the input space and proceed exploring "low-probability" subdomains.

Acknowledgment

I would like to express my deepest gratitude to my advisor, Abhik Roychoudhury (abhik@comp.nus.edu.sg), for his support and guidance throughout this research.

[7]Formula: $input_P \wedge input_{P'} \wedge (output_P \neq output_{P'})$.

References

[1] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 703–711, July 1991.

[2] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence (program testing)," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1402–1411, 1990.

[3] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 438 –444, july 1984.

[4] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[5] D. Qi, H. D. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11, 2011, pp. 278–288.

[6] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *ICSE*, 2009, pp. 56–66.

[7] T. Wang and A. Roychoudhury, "Automated path generation for software fault localization," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05, 2005, pp. 347–351.

[8] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," *Automated Software Engineering, International Conference on*, vol. 0, p. 30, 2003.

[9] M. Böhme and A. Roychoudhury, "Comprehensive test suite augmentation," National University of Singapore, https://dl.comp.nus.edu.sg/dspace/handle/1900.100/3543, Tech. Rep., November 2011.

[10] A. Podgurski and L. A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 965–979, September 1990.

[11] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '04, 2004, pp. 432–448.

[12] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03, 2003, pp. 308–318.

[13] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," in *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '94, 1994, pp. 80–94.

[14] D. Qi, A. Roychoudhury, and Z. Liang, "Test generation to expose changes in evolving programs," in *ASE*, 2010, pp. 397–406.

[15] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional may-must program analysis: unleashing the power of alternation," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '10, 2010, pp. 43–56.

[16] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "express: guided path exploration for efficient regression test generation," in *ISSTA*. ACM, 2011, pp. 1–11.

[17] J. M. Voas, "Pie: A dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, pp. 717–727, 1992.

[18] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *ASE*, 2008, pp. 218–227.

[19] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *ESEC/SIGSOFT FSE*, 2011, pp. 212–222.

[20] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu, "Differential symbolic execution," in *SIGSOFT FSE*, 2008, pp. 226–237.

[21] M. MirzaAghaei, "Automatic test suite evolution," in *SIGSOFT FSE*, 2011, pp. 396–399.

[22] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *ESEC/SIGSOFT FSE*, 1999, pp. 303–321.