REGRESSION TESTING OF MUTATION BASED UNIT TEST CASES IN OOP

[1]*T.SHYAM KUMAR* [2]*M.PRIYANKA*
[1]Associate Professor and HOD-IT, Aurora College,
Bandlaguda, Hyderabad, India.
[2]M.Tech Scholar, Aurora College, Bandlaguda,
Hyderabad, India.

ABSTRACT- *In this paper, we present TEST, an approach that automatically generates unit tests for object-oriented classes based on mutation analysis. By using mutations rather than structural properties as coverage criterion, we not only get guidance in where to test, but also what to test for. This allows us to generate effective test oracles, a feature raising automation to a level not offered by traditional tools.*

This process requires a deep understanding of the source code and is a nontrivial task. Automated test generation can help in covering code (and thus hopefully detecting mutants), but even then, the tester still needs to assess the results of the generated executions—and has to write hundreds or even thousands of oracles.

The test cases of an existing test suite are executed on a program version (mutant) containing one such fault at a time in order to see if any of the test cases can detect that there is a fault. A mutant that is detected as such is considered dead and of no further use. A live mutant, however, shows a case where the test suite potentially fails to detect an error and therefore needs improvement. There are two main problems of mutation analysis: First, the sheer number of mutants and the effort of checking all tests against all mutants can cause significant computational costs. Second, equivalent mutants do not observably change the program behavior or are even semantically identical, and so there is no way to possibly detect them by testing.

1. INTRUDUCTION

Regression testing Since the introduction of mutation analysis, a number of optimizations has been proposed to overcome possible performance problems, and heuristics can identify a small fraction of equivalent mutants at the end of the day, however, detecting equivalent mutants is still a job to be done manually. If the mutant method/constructor is executed but the mutated statement it is not, then the fitness specifies the distance of the test case to executing the mutation; again, we want to minimize this distance. This basically is the prevailing approach level and branch distance measurement applied in a search-based test data generation. The approach level describes how far a test case was from the target in the control flow graph when it deviated course. This is usually measured as the number of unsatisfied control dependencies between the point of deviation and the target, and is 0 if all control dependent branches are reached. The branch distance estimates how far the branch at which execution diverged from reaching the mutation is from evaluating to the necessary outcome. In addition, one can use the necessity conditions definable for different mutation operators to estimate the distance to an execution of the mutation that infects the state (necessity distance). To determine these values, the test case has to be executed once on the unmodified software.

Data theft attacks are increase the volume of the attacker is a intended to do harm insider. This is considered as one of the top effective threats to cloud computing by the Cloud privacy Alliance. While most Cloud computing users are well-aware of this effective threat, they are left only with .If the mutation is executed, then we want the test case to propagate any changes induced by the mutation such that they can be observed. Traditionally, this consists of two conditions: First, the mutation needs to infect the state (necessity condition). This condition can be

formalized and integrated into the fitness function (see above). In addition, however, the mutation needs to propagate to an observable state (sufficiency condition)—it is difficult to formalize this condition. Therefore, we measure the impact of the mutation on the execution; we want this impact to be as large as possible. Quantification of the impact of mutations was initially proposed using dynamic invariants. The more invariants of the original program a mutant program violates, the less likely it is to be equivalent. A variation of the impact measurement uses the number of methods with changed coverage or return values instead of invariants.

2. RELATED REVIEW

Effectiveness of Debugging When Random Test Cases Are Used

Automatically generated test cases are usually evaluated in terms of their fault revealing or coverage capability. Beside these two aspects, test cases are also the major source of information for fault localization and fixing. The impact of automatically generated test cases on the debugging activity, compared to the use of manually written test cases, has never been studied before. In this paper we report the results obtained from two controlled experiments with human subjects performing debugging tasks using automatically generated or manually written test cases. We investigate whether the features of the former type of test cases, which make them less readable and understandable (e.g., unclear test scenarios, meaningless identifiers), have an impact on accuracy and efficiency of debugging. The empirical study is aimed at investigating whether, despite the lack of readability in automatically generated test cases, subjects can still take advantage of them during debugging.

We focused on modeling user search behavior to reveal an attacker's malicious intent. Automatic test case generation is an important area of software testing. The scientific community has reserved a great attention to test generation techniques, which are now available for many popular frameworks and programming languages. In this paper, we analyse the role that the length of test sequences plays in testing software with internal state. In particular, we concentrate on the branch coverage criterion, because it is one of the most common criteria in the literature. To obtain high coverage, a common practice is to apply a first phase of random testing, followed by more sophisticated techniques aimed to cover the remaining branches (control flow analysis can be used to reduce their number by considering their dependences, e.g. nested branches).

3. RELATED WORK

To assess the quality of test suites, mutation analysis seeds artificial defects (mutations) into programs; a non-detected mutation indicates a weakness in the test suite. We present an automated approach to generate unit tests that detect these mutations for object-oriented classes.

This has two advantages: First, the resulting test suite is optimized towards finding defects rather than covering code. Second, the state change caused by mutations induces oracles that precisely detect the mutants. Evaluated on two open source libraries, our μtest prototype generates test suites that find significantly more seeded defects than the original manually written test suites.

4. PROPOSED WORK

Since the introduction of mutation analysis, a number of optimizations have been proposed to overcome possible performance problems, and heuristics can identify a small fraction of equivalent mutants at the end of the day, however, detecting equivalent mutants is still a job to be done manually. A recent survey paper concisely summarizes all applications and optimizations that have been proposed over the years. Javalanche is a tool that incorporates many of the proposed optimizations and made it possible to apply mutation analysis to software of previously unthinkable size. In addition, Javalanche alleviates the equivalent mutant problem by ranking mutants by their impact: A mutant with high impact is less likely to be equivalent, which allows the tester to focus on those mutants that can really help to improve a

test suite. The impact can be measured, for example, in terms of violated invariants or effects on code coverage.

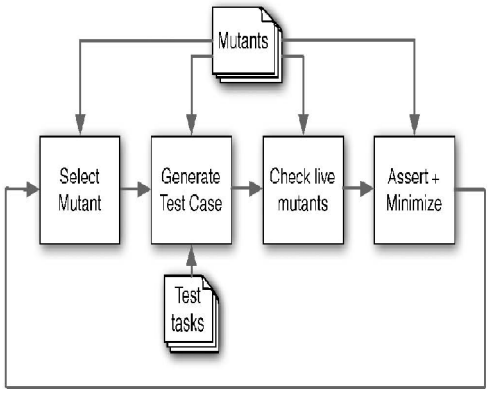

Figure : System Architecture

5 SYSTEM MODULES

Distance to Mutation

If the mutant method/constructor is executed but the mutated statement it is not, then the fitness specifies the distance of the test case to executing the mutation; again, we want to minimize this distance. This basically is the prevailing approach level and branch distance measurement applied in a search-based test data generation. The approach level describes how far a test case was from the target in the control flow graph when it deviated course. This is usually measured as the number of unsatisfied control dependencies between the point of deviation and the target, and is 0 if all control dependent branches are reached. The branch distance estimates how far the branch at which execution diverged from reaching the mutation is from evaluating to the necessary outcome. In addition, one can use the necessity conditions definable for different mutation operators to estimate the distance to an execution of the mutation that infects the state (necessity distance). To determine these values, the test case has to be executed once on the unmodified software.

6. CONCLUSION

Mutation analysis is known to be effective in evaluating existing test suites. In this paper, we have shown that mutation analysis can also drive automated test generation.

The main difference between using structural coverage and mutation analysis to guide test generation is that a mutation does not only show where to test, but also helps in identifying what should be checked for.

In our experiments, this results in test suites that are significantly better in finding defects than the (already high quality) manually written ones.

The advent of automatic generation of effective test suites has an impact on the entire unit testing process: Instead of laboriously thinking of sequences that lead to observable features and creating oracles to check these observations, the tester lets a tool create unit tests automatically and receives two test sets: one revealing general faults detectable by random testing, the other one consisting of regular unit tests. In the long run, finding bugs could thus be reduced to the task of checking whether the generated assertions match the intended behavior.

7. REFFERENCES

[1]S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege, “A Systematic Review of the Application and Empirical Investigation of Search-Based Test-Case Generation,” IEEE Trans. Software Eng., vol. 36, no. 6, pp. 742-762, Nov./Dec. 2010.

[2.] J.H. Andrews, L.C. Briand, and Y. Labiche, “Is Mutation an Appropriate Tool for Testing Experiments?” Proc. 27th Int’l Conf. Software Eng., pp. 402-411, 2005.

[3.] J.H. Andrews, A. Groce, M. Weston, and R.G. Xu, “Random Test Run Length and Effectiveness,” Proc. IEEE/ACM 23rd Int’l Conf. Automated Software Eng., pp. 19-28, 2008.

[4]. J.H. Andrews, S. Haldar, Y. Lei, and F.C.H. Li, “Tool Support for Randomized Unit Testing,” Proc. First Int’l Workshop Random Testing, pp. 36-45, 2006.

[5]. A. Arcuri, “It Does Matter How You Normalise the Branch Distance in Search Based

Software Testing," Proc. Third Int'l Conf. Software Testing, Verification and Validation, pp. 205-214, 2010.

[6]. A. Arcuri, "Longer Is Better: On the Role of Test Sequence Length in Software Testing," Proc. Third Int'l Conf. Software Testing, Verification and Validation, pp. 469-478, 2010.

[7]. A. Arcuri and L. Briand, "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering," Proc. IEEE Int'l Conf. Software Eng., pp. 1-10, 2011.

[8] A. Arcuri and X. Yao, "Search Based Software Testing of Object- Oriented Containers," Information Sciences, vol. 178, no. 15, pp. 3075-3095, 2008.

[9]. K. Ayari, S. Bouktif, and G. Antoniol, "Automatic Mutation Test Input Data Generation via Ant Colony," Proc. Ninth Ann. Conf. Genetic and Evolutionary Computation, pp. 1074-1081, 2007.

[10]. M. Boshernitsan, R. Doong, and A. Savoia, "From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing," Proc. Int'l Symp. Software Testing and Analysis, pp. 169-180, 2006.

[11]. L. Bottaci, "A Genetic Algorithm Fitness Function for Mutation Testing," Proc. Int'l Workshop Software Eng. Using Metaheuristic Inovative Algorithms, a Workshop at 23rd Int'l Conf. Software Eng., pp. 3-7, 2001.

[12]. V. Chvatal, "A Greedy Heuristic for the Set-Covering Problem," Math. Operations Research, vol. 4, no. 3, pp. 233-235, 1979.

Authors

M.PRIYANKA (M.Tech Scholar), Aurora Bandlaguda. My area of interest and research is in software engineering

GUIDE: T. SHYAM KUMAR is working as Head, Department of Information Technology Aurora Bandlaguda Hyderabad, and India. He has received M.Tech. (Computer Science and Technology) from JNTUH. Presently, he is a Research Scholar of JNTUH Hyderabad. He has published and presented more than 10 Research and technical papers in International Conferences and National Conferences. His main research interests are Software Engineering, Software Metrics, Software Quality and OOD.