Lakshman Madhav Kollipara

CS562_Final Report

Project Report

CS562_Applied Software Engineering

Alex Groce

My project report is organized as follows: I have started working on tarantula and went ahead but due to some issues faced and lack of time, I have to come out of it and I proceeded with working on papers published by well know authors on software testing.

So, In first part, I am discussing what I learnt about tarantula, its usefulness and problems faced while working on it and other Fault Localization Techniques existing earlier to Tarantula.

Second part consists of what I learnt about Mutation Testing and some new concepts and study done by Andreas Zeller on Mutation Testing

Tarantula:

I start with a statement, “Fault causer error leading to a failure”. Fault is an incorrect part of program, which is executed and causes error. This error propagates through the program and causes failure. So, fault causes failure which is known as causality.

So, to find the causes of failure, we use fault localization techniques. Tarantula is one of the fault localization approach by Jones & Harrold. Tarantula is one of the fault localization techniques used for automated debugging.

Compared to other fault localization techniques like Set Union, Set Intersection, Nearest Neighbors, Cause transition, Tarantula is best option.

Tarantula utilizes information like pass/fail information about each test case, the entities that were executed by each test case (e.g., statements, branches, methods), and the source code for the program under test. The intuition behind Tarantula is that entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases.

Tarantula takes the number of passed/failed test cases and source code with coverage as input and gives the suspiciousness value of each line in source code as output.

The suspiciousness is a value between 0 and 1. 0 says that that particular statement in code is less likely to be faulty whereas suspiciousness of 1 says that that particular statement is faulty code.

Suspiciousness is calculated based on below formula.

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$

Lakshman Madhav Kollipara

CS562_Final Report

Let ‘e’ be the statement in code,

failed(e) is the number of failing tests which cover ‘e’.

passed(e) is the number of passed test cases which cover ‘e’

Working with Tarantula:

While implementing tarantula, I consulted few people, who are working on tarantula; Amin Alipour and Arpit Christi. As per their suggestions, I have went through jones’s paper and worked on tarantula Java code by Jones.

The tarantula suspiciousnesscalculation takes a test suite pass/fail details in matrix format as input. The coverage matrix is a two dimensional matrix which consists of test number, statements which are covered by test. I have prepared a test suite with 5 test cases each consisting of one full round of dominion game with calling all top level APIs at least once I have 5 test cases with different num_Players and different set of kingdomCards and seed. I found difficulties in constructing a matrix from the coverage XML file of python.

Then I tried with source code of suspiciousness calculation which takes input of coverage file using gcov in C and gives the output. But The problem I faced is the coverage run and report in python gives only whether that particular line has been executed or not but not how many times each statement has been executed. So, I am unable to figure out how to get the count of how many times each statement in code has been executed because I am not that familiar with python.

I tried running the same code with coverage output of gcov in C and it is working fine. You can see the code and output in Tarantula folder. So, finally I gave up the idea of applying tarantula on python and proceed with some conceptual testing techniques because of time constraint and in availability of faculty or people to clarify doubts.

Pseudocode for tarantula using gcov in C:

```
def run_tarantula(tests):
    src_file = tests.get_gcov_file()
    cmd = "gcov " + src_file
    print "Running '" + cmd + "'"
    if subprocess.call(["gcov", src_file]) == 0:
        print "Gcov returned success!"
    else:
        print "Gcov initial run failed!"
        sys.exit(1)
```

```
result_file_name = src_file + ".gcov"
tarantula_lines = {}
gcov_lines = interpret_gcov_file(result_file_name)
for line in gcov_lines:
    tarantula_lines[line[0]] = taran_line(line[2], line[0], exec_count=line[1])
totalfailed = 0
totalpassed = 0
while tests.next_test():
    test_result = tests.run_test()
    if test_result:
        totalpassed += 1
    else:
        totalfailed += 1
    subprocess.call(["gcov", src_file])
    gcov_lines = interpret_gcov_file(result_file_name)
    for line in gcov_lines:
        if line[1] > tarantula_lines[line[0]].exec_count:
            tarantula_lines[line[0]].set_exec_count(line[1])
            if test_result:
                tarantula_lines[line[0]].inc_passed()
            else:
                tarantula_lines[line[0]].inc_failed()
for key in tarantula_lines:
    out_str = tarantula_lines[key].get_cmdline_str(totalpassed, totalfailed)
    if out_str != "":
        print out_str
```

Other Fault Localization Approaches:

Set Union and Set Intersection:

The Set-union technique computes a set by removing the union of all statements executed by all passed test cases from the set of statements executed by a single failed test case.

P- given set of passing test cases

P_i - individual passed test cases in P

F - single failing test case

E_p - set of coverage entities executed by each p

E_f – set of coverage entities executed by f

The union model gives

$$E_{initial} = E_f - \bigcup_{p \in P} E_p$$

The Set-intersection technique computes the set difference between the set of statements that are executed by every passed test case and the set of statements that are executed by a single failing test case. A set of statements is obtained by intersecting the set of statements executed by all passed test cases and removing the set of statements executed by the failed test case. The intuition is to give the statements that were neglected to be run in the failed test case, but were run in every passed test case. We can express the Set-intersection as

$$E_{initial} = \bigcap_{p \in P} E_p - E_f$$

The resulting set $E_{initial}$ for each of these two techniques defines the entities that are suspected of being faulty. In searching for the faults, the programmer would first inspect these entities.

Renieris and Reiss suggest a technique that provides an ordering to the entities based on the system dependence graph. Under this ranking technique, nodes that correspond to the initial set of entities are identified; they call these blamed nodes. A breadth-first search is conducted from the blamed nodes along dependency edges in both forward and backward directions. All nodes that are at the same distance are grouped together into a single rank. Every node in a particular rank is assigned a rank number, and this number is the same for all constituent nodes in the rank. Given a distance d, and a set of nodes at that distance S(d), the rank number that is assigned to every node in S(d) is the size of every set of nodes at lesser distances plus the size of S(d). Using the size of the rank plus the size of every rank at a lesser distance for the rank number gives the maximum number of nodes that would have to be examined to find the fault following the order specified by the technique.

The problem with set-based coverage techniques are most faulty statements are executed by some combination of both passed and failed test cases. However, when using set operations on coverage-based sets, the faulty statement is often removed from the resulting set of statements to be considered.

Lakshman Madhav Kollipara

CS562_Final Report

Nearest Neighbor:

Renieris and Reiss addressed the issue of tolerance for an occasional passed test case executing a fault with their Nearest-Neighbor technique. Rather than removing the statements executed by all passed test cases from the set of statements executed by a single failed test case, they selectively choose a single best passed test case for the set difference. By removing the set of statements executed by a passed test case from the set of statements executed by a failed test case is similar to set union approach, but has a specific technique for specifying which passed test case to use for this set difference. They choose any single failed test case and then find the passed test case that has coverage that is most similar to the coverage of the failed test case. Utilizing these two test cases, they remove the set of statements executed by the passed test case from the set of statements executed by the failed test case. The resulting set of statements is the initial set of statements from which we should start her search for the fault.

Renieris and Reiss defined two measures for the similarity of the coverage sets between the passed and failed test cases.

Binary Distancing:

Computes the set difference of the set of statements covered by the chosen failed test case and the set of statements covered by a particular passed test case. It is defined as either (1) the cardinality of the symmetric set difference of the statements executed by each of the passing and failing test cases, or (2) the cardinality of the asymmetric set difference between the set of statements executed by the failed test case and the set of statements executed by the passed test case.

Permutation Distancing:

In this measure, for each test case, a count is associated with each statement or basic block that records the number of times it was executed by the test case. The statements are then sorted by the counts of their execution. The permutation distance measure of two test cases is based on the cost of transforming one permutation to the other. After an arbitrary failed test case is chosen, the distance value is computed for every passed test case. The passed test case that has the least distance is chosen. They then remove the set of statements executed by this passed test case from the set of statement executed by the failed test case.

The problem with this approach is, This approach is more dependent on test suite which means this doesn't work well if we have no more similar test cases with the failing test case or what if there is no passed test case at all.

Cause Transition:

Cleve and Zeller's Cause-Transitions technique performs a binary search of the memory states of a program between a passing test case and a failing test case. The Cause-Transitions technique defines a method to automate the process of making hypotheses about how state changes will affect output. In this technique, the program under test is stopped in a symbolic debugger using a breakpoint—for both a passed test case and failed test case. Part of the memory state is swapped between the two runs and then allowed to continue running to termination. The memory that appears to cause the failure is narrowed down using a technique much like a binary search with iterative runs of the program in the symbolic debugger. This narrowing of the state is iteratively performed until the smallest state change that causes the original failure can be identified. This technique is repeated at several program points to find the flow of the differing states causing the failure throughout the lifetime of each run. These program points are then used as the initial set of points from which to search for the fault. After this set of program points has been defined, they are specified as the initial set of statements that the programmer uses to search for the faults.

Among all 4 Fault Localization Methods, In terms of Effectiveness i.e the percentage of the program that need not be examined to locate the fault, the Set-intersection techniques perform the worst, followed by Set-union, then Nearest Neighbor using binary distancing, then Nearest-Neighbor using permutation distancing, then the Cause-Transitions using different ranking strategies and then the best result is achieved by the fully automatic Tarantula technique.

The computational time for these techniques and Tarantula is similarly small. The time required by the Tarantula technique for computation is quite small. The I/O cost should also be similar for these techniques. The Nearest-Neighbor technique needs to read in all coverage information for all passing test cases to determine which passing test case will be chosen as the "nearest" one to the failing test case used which requires more time. Similarly for the Set-union and Set-intersection techniques, coverage information for all passing test cases and one failing test case must be read.

The future work in this field can be by applying Tarantula on our harness maker coverage tools and writing a code which creates a coverage matrix when coverage xml file is given.

References:

- J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic FaultLocalization Technique," in Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering, pp. 273-282, Long Beach, California, USA, December, 2005
- H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In Proceedings of IEEE Software Reliability Engineering, pages 143–151, 1995.
- M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In Proceedings of the International Conference on Automated Software Engineering, pages 30–39, Montreal, Quebec, October 2003

- H. Cleve and A. Zeller. Locating causes of program failures. In Proceedings of the International Conference on Software Engineering, pages 342–351, St. Louis, Missouri, May 2005.

Mutation Testing:

What is Mutation Testing?

The simplest way of explaining Mutation testing is, It is a syntax-based coverage checking technique where we generate syntactic mutants of original program and run the test suit on those programs and check how many mutants will the test suit detect. The more mutants detected, the more efficient your test suit is. Detecting all mutants is hard to achieve.

There are two type of Mutants: Non Equivalent mutants and Equivalent Mutants:

The Non-equivalent mutants are manually introduced in the program which can be detected by efficient test suite. The mutations which keep the program semantics unchanged and cannot be detected by any test suite are known as Equivalent Mutants which must be weeded out manually.

I tried Mutation testing by changing trash_flag in discard_Card, changing equality symbols for update_Coins and few others in card_Effect. Then I ran my random tests which I created during assignment2, My test suite found only 2 out of 5 total mutants. Of course, there are many bugs in my dominion (Can see bug report of pranjal on mydominion) and my tests are also not that effective. I have fixed so many bugs in dominion and made it reliable up to some extent.

My test coverage of dominion is around 60-70% which is because my playcard has wrong implementation. So, prior to mutation testing I need to make sure that my dminion is working perfectly fine and it is calling all kingdomcards properly in cardEffect without any compilation and logical errors. Then Mutation would be interesting as I can rely on my code and create multiple mutants.

I am discussing about paper published on Mutation Testing by Andreas Zeller.

Mutation-driven Generation of Unit Tests and Oracles (By Andreas Zeller & Gordon Fraser)**:**

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. So, they came up with an automated approach to generate unit tests that detect these mutations. The advantages of this approach are, the resulting test suite is optimized towards finding defects rather than covering code and the state change caused by mutations induces oracles that precisely detect the mutants.

If a mutant is not detected, then the test suite is not effective and in most cases that either a new test case should be added, or that an existing test case needs a better test oracle. Improving test cases after mutation analysis usually means that the tester has to go back to the drawing-board and design new test cases, taking the feedback gained from the mutation analysis into account which actually is a tedious task and requires lot of time to understand source code. Though we automate the test generation we still need time to assess the results of the generated executions to write hundreds of oracles. So, they came up with a proto type called ‘µtest’, which automatically generates unit tests based on mutation analysis. µtest simplifies the act of testing to checking whether the generated assertions are valid by creating oracles along with test cases.

They implemented µtest as an extension to the Javalanche mutation system, which is an open source framework for mutation testing Java programs with a special focus on automation, efficiency, and effectiveness. Javalanche is built for efficiency from the ground up, manipulating byte code directly and allowing mutation testing of programs that are several orders of larger magnitude.

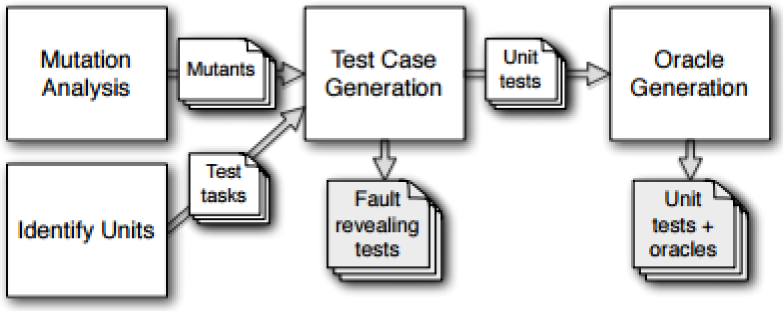

Figure 3: The μTEST process: Mutation analysis, unit partitioning, test generation, oracle generation.

As shown in the above picture, Architecture of µtest process has Mutant Analysis, Test case Generation and Oracle generation. We will discuss about each of them.

Mutation Analysis:

The idea of mutation analysis is to seed artificial faults (Mutants). The test cases of an existing test suite are executed on a program versions containing mutants 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. If the test suite potentially fails to detect a live mutant, that test suite needs an improvement.

Test Case Generation:

Using Automated test case generation, we can generate test cases from models or source code, using different test objectives such as coverage criteria. Here they considered only white-box techniques that require no specifications and consider the control-flow of the program and try to cover as many aspects as possible. They used generation of random unit tests implemented in Randoop. Although there is no guarantee of reaching certain paths, random testing can achieve relatively good coverage with low computational needs.

Test case Generation for Mutant Testing:

µtest uses a genetic algorithm to breed method/constructor call sequences that are effective in detecting mutants. The test case generation to kill mutants is natural extension of mutation analysis which adapt constraint based testing. The idea is similar to test generation using symbolic execution and constraint solving, but in addition to the path constraints, each mutation adds a condition that needs to be true such that the mutant affects the state.

Oracle Generation:

By comparing executions of a test case on a program and its mutants, we generate a reduced set of assertions that is able to distinguish between a program and its mutants.

Orstra (is a tool which augments an automatically generated unit-test suite with regression oracle checking) generates assertions based on observed return values and object states and adds assertions to check future runs against these observations and DiffGen (a coverage-based test generation tool to generate test inputs for covering the added branches to expose behavioral differences) extends the Orstra approach to generate assertions from runs on two different program versions. This is similar to our µtest approach, although we do not assume two existing different versions of the same class but generate many different versions by mutation, and are therefore not restricted to a regression testing scenario.

Killing Mutants:

A mutant is only detected if there is an oracle that can identify the misbehavior that distinguishes the mutant from the original program. Consequently, mutation-based unit tests need to add assertions as test oracles such that the mutants are detected. Some types of assertions (e.g., assertions on primitive return values) can be easily generated, as demonstrated by existing testing tools. This, however, is not so easy for all types of assertions, and the number of possible assertions in a test case typically exceeds the number of statements in the test case by far. Mutation testing helps in precisely this matter by suggesting not only where to test, but also what to check for. After the test case generation process we run each test case on the unmodified software and all mutants that are covered by the test case, while recording traces with information necessary to derive assertions. We use the following types of assertions;

Primitive assertions make assumptions on primitive (i.e., numeric or Boolean) return values of method calls. Comparison assertions compare objects of the same class with each other. Comparison of objects across different execution traces is not safe because it might be influenced by different memory addresses, which in turn would influence assertions; therefore we compare objects within the same test case. Inspector assertions call inspector methods on objects to identify their states. An inspector method is a method that takes no parameters, has no side-effects, and returns a primitive data type. Field assertions are a variant of inspector assertions and compare the public primitive fields of classes among each other directly. String assertions compare the string representation of objects by calling the toString method.

Generating Assertions for Test cases:

To generate assertions for a test case we run it against the original program and all mutants using observers to record the necessary information: An observer for primitive values records all observed return and field values, while an inspector observer calls all inspector methods on existing objects and stores the outcome, and a comparison observer compares all existing objects of equal type and again stores the outcome. After the execution the traces generated by the observers are analyzed for differences between the runs on the original program and its mutants, and for each difference an assertion is added. At the end of this process, the number of assertions is minimized by tracing for each assertion which mutation it kills, and then finding a subset for each test case that is sufficient to detect all mutations that can be detected with this test case. This is an instance of the NP-hard minimum set covering problem, and we therefore use a simple greedy heuristic. The heuristic starts by selecting the best assertion, and then repeatedly adds the assertion that detects the most undetected mutants. To give preference to other types of assertions, string assertions are only added at the end of this procedure to cover those mutants that can only be covered by string assertions.

Conclusion:

Though, there is nothing to do much with this approach in dominion, because this is proposed for Object oriented systems mainly like Java. If we have a java version of dominion, then using Javalanche, we can perform this approach. It requires some knowledge on Javalanche as mentioned above. But this approach looks interesting because the effectiveness of test-suite is mainly dependent on the number of seeded mutants it kills. Though your test-suite gives you more coverage but failed to detect syntactic changes in your program then it is more likely to ignore real time errors in program. This approach generates test cases and oracles keeping in mind of detecting Non-equivalent mutants by various assertion methods. As per their experimental results, test cases generated by µtest detects 83% of Mutants where as manual test cases detected only 74% of mutants in Joda-Time case study. Joda-Time provides a quality replacement for the Java date and time classes. The design allows for multiple calendar systems, while still providing a simple API and an extensive set of utility functions. Joda-Time is known for its comprehensive set of developer tests, providing assurance of the library's quality, which makes it well suited as a benchmark to compare automatically generated tests with

References:

- Gordon Fraser and Andreas Zeller. Mutation-driven Generation of Unit Tests and Oracles. In Proceedings of fraser-issta-2010, Trento, July 2010.
- R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. Computer, 11(4):34–41, April 1978.
- R. B. Evans and A. Savoia. Differential testing: a new approach to change detection. In ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering,

- K. Taneja and T. Xie. DiffGen: Automated regression unit-test generation. In ASE 2008: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, pages 407–410, 2008
- D. Schuler and A. Zeller. Javalanche: efficient mutation testing for Java. In ESEC/FSE '09: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, p
- C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, pages 815–816, New York, NY, USA, 2007. ACM.
- T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In ECOOP 2006: Proceedings of the 20th European Conference on Object-