Combinatorial Interaction Testing Using Test Case Based Constraints

[1]K.Pavan Kumar, [2]T.S.Shiny Angel
[1]Mtech Student, [2]Assistant Professor
[1]Software Engineering, SRM University , Chennai, India
[1]pavank278@gmail.com, [2]angel_d_leno@yahoo.co.in

***Abstract*— The bigger applications like web servers e.g. Apache, databases e.g. mysql, and application servers e.g. Tomcat are required to be customizable to adapt to particular runtime contexts and application scenarios. One way to support software customization is to provide configuration options through which the behavior of the system can be controlled. But the configuration spaces of modern software systems are too large to test exhaustively. The proposed method called traditional combinatorial interaction testing which samples the covering arrays and the test cases to make the highly configured application or system. In the combinatorial interaction testing generally we generate configuration options and then we apply test cases for each configuration options. That causes masking effect or skipping of the some reliable configuration options. The obtained system is towards to highly configurable system which uses traditional combinatorial Interaction Testing. Traditional Combinatorial Interaction Testing generates test cases with configuration options and uses test case specific constraints and seeding which avoids the masking effect.**

***Index Terms*—Combinatorial interaction testing, configurations options, configuration space, covering array, masking effect.**

I. INTRODUCTION

Test case-aware covering arrays aim to ensure that each test case has a fair chance to test all of its valid t-tuples. To this end, each test case is scheduled to be executed only in configurations which are valid for the test case so that no masking effects can occur. In other words, for a given configuration space model, a t-way test case-aware covering array is a set of configurations, each of which is associated with a set of test cases, indicating the test cases scheduled to be executed in the configuration such that 1)none of the selected configurations violate the system-wide constraint, 2) no test case is scheduled to be executed in a configuration that violates the test case-specific constraint of the test case, and 3) for each test case, every valid t-tuple appears at least once in the set of configurations in which the test case is scheduled to be executed. An example presents a 3-way test case aware covering array created for our hypothetical. Since none of the test case-specific constraints are violated in this covering array, each test case has a chance to test all of its valid 3-tuples; no masking effects caused by test skips can occur.[4]

Test Case-Specific Constraints

We first observed that the test case-specific constraints were encoded in the test oracles of our subject applications, indicating that they were known to the developers. This is important because the more accurate and complete the test case-specific constraints.

Test Case Generation

Research on automated test case generation has resulted in a great number of different approaches, deriving test cases from models or source code, using different test objectives such as coverage criteria, and using many different underlying techniques and algorithms. It has used white-box techniques that require no specifications; naturally, an existing specification can help in both generating test cases and can serve as test oracle.

Oracle Generation

In the context of regression testing, automated synthesis of assertions is a natural extension of test case generation. Randoop allows annotation of the source code to identify observer methods to be used for assertion generation. Orstra generates assertions based on observed return values and object states and adds assertions to check future runs against these observations. A similar approach has also been adopted in commercial tools such as Agitar.

While such approaches can be used to derive efficient oracles, they do not serve to identify which of these assertions are actually useful, and such techniques are therefore only found in regression testing. Eclat can generate assertions based on a model learned from assumed correct executions; in contrast, µtest does not require any existing executions to start with.

II. RELATED WORK

Whole Test Suite Generation

Whole test suite generation proposes the tool to automatically generate test oracles. A common scenario in software testing is therefore that test data is generated, and a tester manually adds test oracles. But this Whole Test Suite Generation tells about the test case and corresponding test oracles generated efficiently. And task of manually analyzing the code for testing is reduced. It keeps the coverage goal and coverage criteria for the code that for the test case is generated. Tester manually generates test oracles. Coverage goals are not independent. Coverage goals are infeasible.Test generation is therefore dependent on the order of coverage goalsIn this project a method is proposed based on the Genetic Algorithm. This project aims covering *all* coverage goals

at the same time. Its effectiveness is not affected by the number of infeasible targets in the code.Coverage goal are effective and efficient.[1]

Genetic Algorithm

In the computer science field of artificial intelligence, a genetic algorithm (GA) is a search heuristic that mimics the process of natural selection. This heuristic (also sometimes called a met heuristic) is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover.

Mutation-driven Generation of Unit Tests and Oracles

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.[2]

The unit tests against which we compared might not be representative for all types of tests (in particular we did not compare against automatically derived tests), but we chose projects known to be well tested. The units investigated had to be testable automatically. Another possible threat is that the tools the project have used or implemented could be defective. Potentially, the generated oracles might be over fitted to the mutants for which they are constructed, but studies have shown that mutation analysis is well suited to evaluate testing techniques.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.

µtest

µtest creates a test case with an oracle that catches this very mutation. LocalDate object var0 is initialized to a fixed value (0, in our case). Line 2 generates a DateTime object with the same day, month, and year as var0 and the local time (fixed to 0, again). The constructor call for var2 implicitly calls the constructor, and therefore var0 and var2 have identical day, month, and year, but differ by 1 millisecond on the mutant, which the assertion detects. By generating oracles in addition to tests, µtest simplifies the act of testing to checking whether the generated assertions are valid, rather than coming up with assertions and sequences that are related to these assertions. If a suggested assertion is not valid, then usually a bug has been found. Summarizing, the contributions of this paper are:

- Mutant-based unit test case generation:
  µtest uses a genetic algorithm to breed method/constructor call sequences that are effective in detecting mutants. Mutant-based 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.
- Impact-driven test case generation:
  To minimize assessment effort, µtest optimizes test cases and oracles towards detecting mutations with maximal impact—that is, changes to program state all across the execution. Intuitively, greater impact is easier to observe and assess for the tester, and more important for a test suite.
- Mutant-based test case minimization:
  Intuitively, the shorter a test case, the easier it is to understand. We minimize test cases by first applying a multi-objective approach that penalizes long sequences during test case generation, and then we remove all irrelevant statements in the final test case.

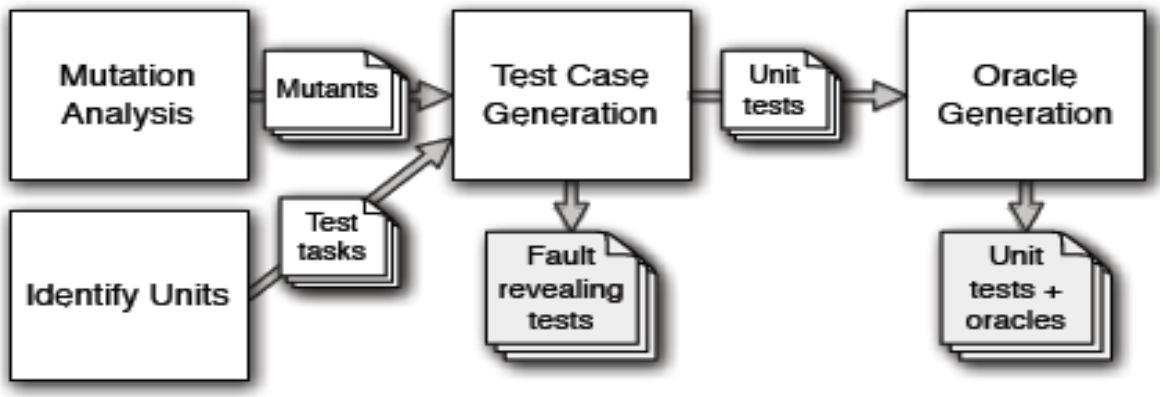

Fig 1

The above diagram shows overall µtest process: The first step is mutation analysis and a partitioning of the software under test into test tasks, consisting of a unit under test with its methods and constructors as well as all classes and class members relevant for test case generation. For each unit a genetic algorithm breeds sequences of method and constructor calls until each mutant of that unit, where possible, is covered by a test case such that its impact is maximized. They minimize unit tests by removing all statements that are not relevant for the mutation or affected by it; finally, and they generate and minimize assertions by comparing the behaviour of the test case on the original software and its mutants.

The authors have implemented µtest as an extension to the Javalanche mutation system. Authors demonstrate its effectiveness by applying it to two open source libraries that have a reputation of being extensively well-tested.Mutation analysis is known to be effective in evaluating existing test suites. Although our µtest experiences are already very promising, there is ample opportunity to improve the results further: For example, previously generated test cases, manual unit tests, or test cases satisfying a coverage criterion could serve as a better starting point for the genetic algorithm.

III. EVOLUTIONARY GENERATION OF WHOLE TEST SUITES

Recent advances in software testing allow automatic derivation of tests that reach almost any desired point in the source code. There is, however, a fundamental problem with the general idea of targeting one distinct test coverage goal at a time: Coverage goals are neither independent of each other, nor is test generation for any particular coverage goal guaranteed to succeed. We present EVOSUITE, a search-based approach that optimizes whole test suites towards satisfying a coverage criterion, rather than generating distinct test cases directed towards distinct coverage goals. The focus of this paper is on comparing the approach "entire test suite" to "one target at the time". Threats to construct validity are on how the performance of a testing technique is defined. We gave priority to the achieved coverage, with the secondary goal of minimizing the length.The proposed method shows the generation of the test case automatically via. test tool generation.[3]

Genetic Algorithm

In the computer science field of artificial intelligence, a genetic algorithm (GA) is a search heuristic that mimics the process of natural selection. This heuristic (also sometimes called a met heuristic) is routinely used to generate useful solutions to optimization and search problems. Genetic algorithms belong to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection.

Proposed Method

This project proposes on the configuration options and test cases for highly configured system. The input to the system with configuration space and test cases based on the test case on specific constraints. The Traditional Combinatorial Interaction Testing is used for the test cases and the configuration options. It overcomes the masking effects. It makes system highly configurable.

Problems addressed

General-purpose, one-size-fits-all software solutions are not acceptable in many application domains. For example, web servers (e.g., Apache), databases (e.g., MySQL), and application servers (e.g., Tomcat) are required to be customizable to adapt to particular runtime contexts and application scenarios. One way to support software customization is to provide configuration options through which the behavior of the system can be controlled. While having a configurable system promotes customization, it creates many system configurations, each of which may need extensive QA to validate. Since the number of configurations grows exponentially with the number of configuration options, exhaustive testing of all configurations, if feasible at all, does not scale well.

Algorithm 1: Maintaining a Separate Configuration Space Model for Each Test Case:

In this first algorithm a different test case is maintained for the every configuration options. A generator is used which generates test configuration options with test cases. The test case specific constraints are made for every test case.

E.g.

Assume configuration options are: o1,o2,o3,o4 and the relevant test cases are as t1,t2,t3;Thus every set of configuration options o1, o2, o3, o4 have some specific constraint for the test cases to run. For t1 only can run in configuration options where o1 is 0. Similarly t2 can run configuration options where o2 is 1.And t3 does not have any constraints. And applies seeding mechanism to overcome the masking effect.

Algorithm 2: Maintaining a Single Configuration Space Model

This algorithm operates in an iterative manner. At each iteration, we select the best configuration and the set of associated test cases which cover the greatest number of previously uncovered t-pairs. The selection is then included in the test case-aware covering array and the t-pairs appearing in the selection are marked as covered. The iterations end when every valid t-pair is covered at least once. Thus above two algorithms are used to minimize the number of configurations included in test case-aware covering arrays.

Algorithm 3: Minimizing Number of Test Runs

However, reducing the number of configurations does not necessarily reduce the number of test run required. The reason for this trade off is test cases to share configurations. It can be avoided by using brute force attack.

IV. IMPLEMENTATION

This project proposes on the configuration options and test cases for highly configured system.The input to the system with configuration space and test cases based on the test case on specific constraints.The Traditional Combinatorial Interaction Testing is used for the test cases and the configuration options.It overcomes the masking effects.It makes system highly configurable. Test case-aware covering arrays aim to ensure that each test case has a fair chance to test all of its valid t-tuples. To this end, each test case is scheduled to be executed only in configurations which are valid for the test case so that no masking effects can occur.
In other words, for a given configuration space model, a t-way test case-aware covering array is a set of configurations, each of which is associated with a set of test cases, indicating the test cases scheduled to be executed in the configuration such that 1) none of the selected configurations violate the system-wide constraint, 2) no test case is scheduled to be executed in a configuration that violates the test case-specific constraint of the test case, and 3) for each test case, every valid t-tuple appears at least once in the set of configurations in which the test case is scheduled to be executed. An example presents a 3-way test case aware covering array created for our hypothetical. Since none of the test case-specific constraints are violated in this covering array, each test case has a chance to test all of its valid 3-tuples; no masking effects caused by test skips can occur.

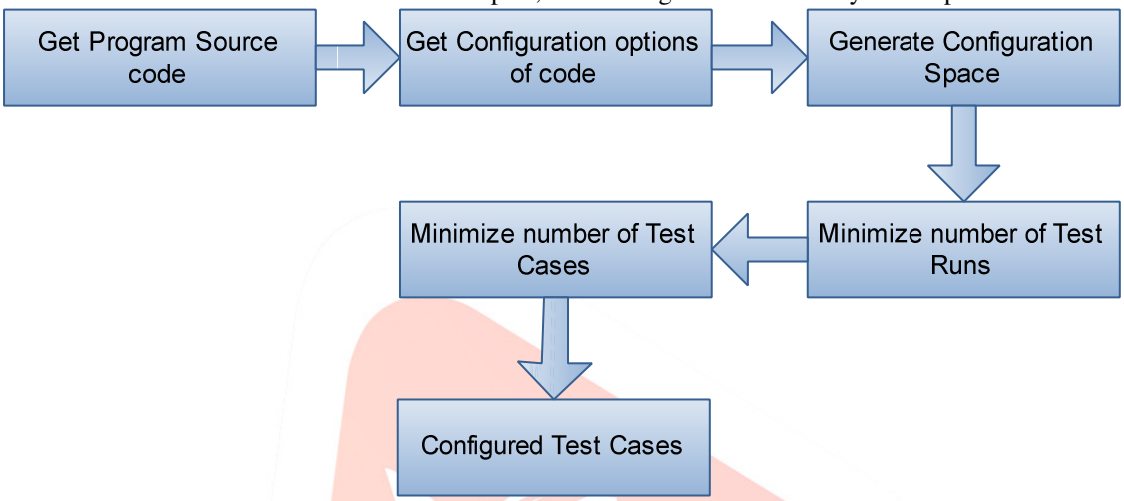

Fig 2

V. CONCLUSION AND FUTURE WORK

Combinatorial interaction testing aim to ensure that each test case has a fair chance to test all of its valid t-tuples. Each test case is scheduled to be executed only in configurations which are valid for the test case so that no masking effects can occur. A test case-specific constraint, on the other hand, applies only to the test case that it is associated with and determines the configurations in which the test case can run. Objective is to make configurable and customizable system by which the behavior of the system can be controlled efficiently. A valuable observation we make is that there is often a tradeoffs between minimizing the number of configurations and minimizing the number of test runs in test case-aware covering arrays. An attempt to reduce one count often increases the other count. These tradeoffs plays an important role in minimizing the total cost of testing, especially when there is a profound practical difference between the cost of configuring the system and the cost of running the test cases. The obtained result will be the highly configured test cases. As a future work, work on cost- and test-case aware covering arrays that support a general cost model in which the overall cost of testing can be specified at the granularity of option settings and test cases.

REFERENCES

[1] Gordon Fraser, Andrea Arcuri," Whole Test Suite Generation", IEEE transactions on software engineering February 2013, Volume: 39, Page no: 276 - 291.
[2] Gordon Fraser, Andreas Zeller, "Mutation-driven Generation of Unit Tests and Oracles", Ieee transaction 2010, volume. 1 Page No 1-9
[3] Gordon Fraser, Andrea Arcuri, "Evolutionary Generation of Whole Test Suites", IEEE transaction 2012,Vol 29, Page No: 639-645.
[4] Cemal Yilmaz , "Test Case-Aware Combinatorial Interaction Testing", IEEE transactions on software engineering, vol. 39, no. 5,may 2013, page no: 684-706
[5] R.C. Bryce and C.J. Colbourn, "Prioritized Interaction Testing for Pair-Wise Coverage with Seeding and Constraints," Information and Software Technology, vol. 48, no. 10, pp. 960-970, 2006.
[6] J. Czerwonka, "Pairwise Testing in the Real World: Practical Extensions to Test-Case Scenarios," Proc. 24th Pacific Northwest Software Quality Conf., pp. 285-294, 2006.
[7] K.-C. Tai and Y. Lei, "A Test Generation Strategy for Pairwise Testing," IEEE Trans. Software Eng., vol. 28, no. 1, pp. 109-111, Jan.2002.
[8] M.B. Cohen, M.B. Dwyer, and J. Shi, "Interaction Testing of Highly Configurable Systems in the Presence of Constraints,"