List of Publications


Published Journals
A Comprehensive Survey of Recent Developments in Software Testing Methodologies

C. Prakasa Rao, P. Govindaraju
Research Scholar, Dept. of Computer Science, S.V. University, Tirupati, India
Assoc. Professor, PEC, Kandukur, India

Abstract: Testing is one of the phases of all process models of software engineering. It is not just one of the most indispensable part of software quality assurance. High reliability of software is expected in the real world as it is more so with complex and machine critical applications. In this paper we provide a comprehensive survey of recent developments in software testing methodologies. Several approaches discussed in this paper include automatic generation of test cases, search based techniques, Just-In-Time quality assurance, static analysis, bad smell detection, early detection of concurrency problems, random testing, integration testing, combinatorial testing, model-based testing, test-driven approaches, dependency-based test case prioritization, state-based testing, adaptive testing and so on. This paper throws light into dependency structures for test case prioritization and test suite generation with minimum size test suites maximum coverage with discussion on empirical studies. The recent methodologies in software testing are focused in this paper besides finding potential gaps for future work.

Keywords: Software testing, test-driven development, search-based methods, automatic software testing

1. Introduction

Software testing is one of the essential parts of software development process. A software test contains the definition of expected output besides the input that is used to execute the program. Many solutions came into existence from automatic testing of software. Test case generation with unit testing, integration testing and other approaches are found in the literature. The test case testing life cycle is presented in Figure 1. As there are phases in system development life cycle, in test life cycles also there are phases involved. Generally these phases are carried out in parallel with the development life cycle. In software testing, the coverage of test cases plays an important role to research all possible defects in the SUT. A common approach in generating test cases is to generate a test case for each coverage goal and combine them into a single test suite as explored in [27]. When a single goal at a time is considered, it generates more test suites or the size of test suite is more. In [12] a representative test suite or whole test suite is generated that will have full coverage besides reducing size of test suites.

There are many approaches to test case generation. They are automatic test case generation approaches [6], [8], [13], [4], [24], and [22], search-based approaches [11], [4] and [11], Just-In-Time quality assurance [35], bad smell detection types and approaches [15], early detection of concurrency problems [22], static analysis for test case generation [19], interaction testing approaches [7], random testing approaches [5], test-driven approaches [34] and [17], dependency based test case prioritization techniques [30], [32], state based testing approaches [31], constraint based testing approaches [14] and [13], model-based approaches [10], model based [31] and [24], state based testing approaches [25] and integration testing [2]. The role of test sequence length in structural coverage of software testing was explored in [7]. There is evidence that there is relationship between test sequence length and structural coverage. Genetic algorithms are used as evolutionary approaches in test case generation as explored in [28], [16], [11] and [12].

In this paper our contribution include the review of literature in finding various methodologies employed for test case generation besides identifying potential research gaps that can help in directing future research. The remainder of the report is structured as follows. Section 2 presents GA based methodologies. Section 3 presents GA based methodologies. Section 3 presents dependency based solutions for software testing. Section 4 presents test-driven software testing approaches. Section 5 provides search-based techniques. Section 6 presents bad smell detection and resolution. Section 7 presents other approaches such as state-based testing, refactoring, architecture based solution, adaptive testing, combinatorial testing, and so on. Section 8 presents the recent research and the gaps in the research that can help in planning for future work. Section 9 concludes the paper besides giving directions for future work.
2. GA Based Methodologies

2.1 Whole Test Suite Generation

Fraser and Arcuri [12] presented a novel approach to generate whole test suite that fulfills all coverage goals besides keeping the size of test suite small. They implemented a tool named EVOSUITE for efficiently testing the whole test suite generation. An evolutionary approach using Genetic Algorithms (GA) is used to achieve this. The solution here is a test suite represented as $T = \{t_1, t_2, t_3, \ldots\}$. Here $t_i$ represents a program that is used to test a part of Software Under Test (SUT). In the same fashion, a test case is treated as sequence of instructions represented as $1 = \{t_1, t_2, t_3, \ldots\}$. The test suite's length is considered as the sum of length of all test cases involved in the test suite. It is represented as $\text{length}(T) = \sum_{i=1}^{n} |t_i|$. The statement or instruction denoted as $t_i$ is of four types namely primitive statement, constructor statement, field statement and assignment statement. Enumeration variables, numeric variables, Strings and Boolean variables come under primitives. The statements used to construct objects come under constructor statements while the statements that are use of public member variables and they are part of a class are known as field statements. The assignment statements are instructions that assign values to variables or array collections. As part of fitness function the notion of branch coverage is used for test criterion. Such fitness value is used to measure how close the test suite is that has maximum coverage. There is similar control mechanism which avoids generating longer test cases. Test suite cross over and test suite mutation are the genetic operators used in the solution [25].

From the experiments conclusions are made such as high coverage can be achieved using whole test suite generation besides producing smaller test suites. Evolutionary algorithms using GA performed better when the tool is compared with other tools that used different approaches to solving the problem. With respect to path coverage the whole test suite generation is compared with other tools such as CUTE [21] and DART [26] and it showed higher performance.

2.2 Other GA Based Methodologies

Baker and Bahli [28] applied mutation testing for testing safety-critical software systems as SUT and experimented on improving the test quality. Program size is one of the characteristics considered for mutation testing. Their experimental results revealed that mutation testing provides measure for test quality. Andrews et al. [16] employed randomized unit testing using genetic algorithms along with a Feature Subset Selection (FSSS) tool for assessing size in GA. This approach was proved to be more useful when compared with search-based approaches. Fraser and Zeller [11] introduced artificial defects called mutations into the program and presented an automated solution for generating test cases. The empirical results revealed that the approach could generate test cases that uncover more defects in the system.

3. Dependency Based Solutions

3.1 Test Suite Prioritization

Fault detection is the main goal of any test suite which has multiple test cases. However, some test cases depend on other test cases. Provided this fact, it is essential to identify dependency structure in order to prioritize test cases. When test cases are prioritized, the resultant functional test suite can produce quality feedback that helps developers to focus on the issues and rectify problems. Haidry and Miller [32] studied the problem of test suite prioritization. They focused on a hypothesis that tells that dependencies among test cases can have their impact on the fault detection rate. The test case prioritization is the process of ordering test cases in order to increase the possibility of fault detection. The
number of defects unearthed in SUT can be called as rate of fault detection. In SUT some interactions should occur prior to other interactions causing dependencies problem [32]. Figure 3 shows a sample dependency structure.

![Sample dependency structure](image)

Figure 3: Sample dependency structure [32]

As shown in Figure 3, the root nodes do not have any dependencies while all other nodes do have dependencies. There are direct dependencies and indirect dependencies. For instance D8 is indirectly dependent on I1 and directly dependent on I3. Dependency is of two types namely open dependency and closed dependency. Open dependency refers to the fact that a test case needs to be executed before another test case but need not be immediately before the test case. The closed dependency says that a test case needs to be executed just before another test case based on the dependency. There might be some dependencies that are combination of closed and open dependencies. Two graph coverage measures are used to know dependency structures. They are known as DSP height and DSP volume. DSP is the acronym for Dependency Structure Prioritization. DSP volume refers to the number of dependencies. DSP height refers to the level of deepest dependencies. DSP volume can be computed as all indirect and direct dependencies of a test case while DSP height is computed as the height of all test paths and considering the one which has longest path. The test cases ordering are done using these two graph measures. Two sets of experiments are made to test both open dependencies and closed dependencies respectively. Figure 4 shows the artifacts collected from real world and the metrics for the SUT.

<table>
<thead>
<tr>
<th>Artifact</th>
<th>Type</th>
<th>Lines of code</th>
<th>Functions</th>
<th>Faults</th>
<th>Tests</th>
<th>Dependencies</th>
<th>Graph density</th>
<th>Unconnected tests</th>
<th>Maximum depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elint1</td>
<td>component</td>
<td>2485</td>
<td>54</td>
<td>72</td>
<td>64</td>
<td>64</td>
<td>0.03175</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>Elint2</td>
<td>component</td>
<td>2487</td>
<td>54</td>
<td>84</td>
<td>64</td>
<td>64</td>
<td>0.03175</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>CSM1</td>
<td>unit</td>
<td>365</td>
<td>10</td>
<td>15</td>
<td>64</td>
<td>64</td>
<td>0.05908</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>CSM2</td>
<td>unit</td>
<td>975</td>
<td>60</td>
<td>60</td>
<td>64</td>
<td>64</td>
<td>0.05908</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>CRM1</td>
<td>component</td>
<td>1875</td>
<td>33</td>
<td>90</td>
<td>64</td>
<td>64</td>
<td>0.08607</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>CRM2</td>
<td>component</td>
<td>1875</td>
<td>33</td>
<td>90</td>
<td>64</td>
<td>64</td>
<td>0.08607</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>MET</td>
<td>system</td>
<td>10.674</td>
<td>270</td>
<td>12</td>
<td>80</td>
<td>80</td>
<td>0.06549</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>C77</td>
<td>component</td>
<td>27246</td>
<td>756</td>
<td>27</td>
<td>548</td>
<td>548</td>
<td>0.02101</td>
<td>161</td>
<td>5</td>
</tr>
<tr>
<td>Bash (+6) system</td>
<td>59,000</td>
<td>105</td>
<td>3.4</td>
<td>106</td>
<td>461</td>
<td>0.00394</td>
<td>460</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

As can be seen in Figure 4, it is evident that the Bash has highest dependencies while the CRM1 and CRM2 have fewest dependencies. These SUTs are used for experiments to know both closed and open dependencies and generate test suites with test case prioritization. The experiments are made to demonstrate the usefulness of test cases in order to increase fault detection rate. Average Percentage of Faults Detected (APFD) is the measure defined in [13] used to know the rate of fault detection. The more its value is the more the rate of fault detection.

![APFD for closed dependencies](image)

Figure 5: APFD for closed dependencies [32]

Volume 3 Issue 10, October 2014

www.ijsr.net

Licensed Under Creative Commons Attribution CC BY
As shown in Figure 5 and Figure 6, it is evident that the many algorithms are employed to test the APFD for closed and open dependencies respectively. There are DSP prioritization methods and other methods that do not use DSP measures. The empirical results revealed that the DSP prioritization methods achieved higher APFD when compared with non-DSP prioritization methods. Both experiments proved that DSP measures yield better performance in test case prioritization and also detection ratio for given SUT. There are three test case prioritization techniques that are close to the approach followed in [32]. They include history - based [36] and model-based [9]. The final one takes information from prior execution cycles; the second one uses human knowledge for the task while the third uses a model of the system for test case prioritization.

3.2 Cyclic Dependencies and Quality of Software

Oyetayo et al. [30] studied cyclic dependencies in SUT and the quality of the software. They made experiments on the object-oriented metrics on cyclic dependencies to reduce the errors with error-proneness. The results revealed that the cyclic components in software caused more defects in the system. This will have influence in software testing and maintenance. Rectifying such components is required in order to improve the quality of software. Another observation is that software complexity adds to the error-proneness.

As seen in Figure 8, it is evident that there is a high rate of defective classes in cyclic groups. For instance Apache Camel exhibits 90% defects at class level and 83% at package level. This way other products and their defective components are presented in Figure 8.

4. Test-Driven Software Testing Approaches

Rafique and Mišić [34] studied the impact of test-driven development (TDD) on productivity and code quality. Developer's task size, test-driven approach has significant influence on the quality of software. TDD has positive effect on quality of software. Meta - analytical techniques were used to know the effectiveness of TDD in software quality. However, the productivity of TDD is insinuative as it needed further research efforts. Wilkerson et al. [17] presented two approaches for software testing. The first approach is code inspection while the second approach is test-driven development. As far as reduction of defects is concerned, the code inspection has more advantages. However, it proved to be expensive. When compared to traditional programming methods, the TDD approach has no significant improvement over the code inspection approach. Code inspection proved to be more effective than TDD.

5. Search Based Testing Approaches

In [20] meta-heuristic algorithms were applied for search based testing. The algorithms were integrated with a tool named AUSTIN (Augmented Search Based Testing). The tool is meant for structural data generation so as to cover all branches in SUT. A genetic symbolic execution was used along with path-based approaches. The climbing search algorithm is the first level, that is, it is one of history-based algorithms and also the neighborhood search, which is search based and is followed by the evolutionary algorithm. The evolutionary algorithm was used with the help of test cases built in C language. The Hill Climbing Algorithm (HCA) is based on the local search. The Greedy Algorithm (GA) is used in the search based testing. The algorithms include Genetic Algorithm (GA), Hill Climbing Algorithm (EA), Hill Climbing (HC) and Random Search (RS). The performance of these algorithms was tested under different lengths of test sequences. The empirical results revealed that different algorithms provided performance differently based on SUT. However, a fact proven is that the length of test sequence

Volume 3 Issue 10, October 2014

www.ijsr.net

Licensed Under Creative Commons Attribution CC BY

Figure 8: Defective components in cyclic group in class and package respectively [30]
test data generation. With respect to oiled algorithms, fitness function is used to guide the outcomes in the code. Towards this end, they presented a sequence of steps that can be used to achieve the desired results.

6. Saving Effort through Bad Smell Detection

Potential problems can exist in SUT. The sign of such problems can be called as bad smell. Liu et al. [15] explored the concept of scheduling bad smell detection for resolving issues in the code. Towards this end, they presented a sequence of steps that can be used to achieve the desired results.

![Diagram of procedure for detection resolution of bad smells](image)

**Figure 10:** Procedure for detection resolution of bad smells

As can be seen in Figure 10, it is evident that there is certain procedure to be followed to detect various kinds of bad smells such as duplicate code, long method, large class, long parameter list, unused class, method, field, primitive obsession, feature envy etc. Pair wise resolution sequences are constructed in order to detect and resolve bad smells with ease. This also reveals potential relationships among bad smells.

Different kinds of bad instances of a specific smell are detected kind of bad smell are and resolved one after another. The detection of one kind of bad smell can help in detecting other kinds of bad smells.

![Diagram of pair wise resolution sequences](image)

**Figure 11:** Illustrates pair wise resolution sequences [15]

Schematic overview of the pair wise resolution sequences is presented in Figure 11 that help in finding dependencies and relationships so as to detect bad smells easily. Commonly occurring bad smells in source code can be detected in this manner. When this knowledge is applied in testing software products, it can greatly help in identifying bad smells and generate a comprehensive report which can guide developers taking necessary steps [15].

7. Other Testing Mechanisms

Kamei et al. [35] focused on Just-In-Time quality assurance by unearthing potential errors in code in the early stages of software development lifecycle. This solution overcomes the problems of „traditional“ quality assurance approaches. The benefits of this approach include coarse-grained prediction of bugs, ability to identify relevant errors, and late predictions. Andrews [16] focused on randomized unit testing using GAs. Goues et al. [8] presented a generic method that detects problems in software and repair it automatically. Their method is named „Selfheal“ which is based on GA, Delta-debugging and restructured differential techniques in order to repair software. Shonsha et al. [22] presented a solution for early detection of concurrency issues such as deadlocks and starvation in software using UML. Modeling. Yilmaz [7] presented combinatorial interaction testing which covers arrays and collections which is an easy case aware. Various configuration space models have explored for testing SUT. Masbah et al. [3] proposed a novel method for automatic testing of AJAX-based modern applications. Fault detection was achieved using DOM tree variants that can be used as test oracles. Their approach is known as invariant based automatically testing that supports plug-in for scalable and expandable solution.

Arcuri et al. [5] studied random testing. Their empirical study reveals the relationship between the random testing and quality of SUT. Schaffer et al. [23] focused on accessibility and naming problems with respect to
Genetic Algorithm for Automatic Generation of Representative Test Suite for Mutation Testing

C. Prakasa Rao*
Research Scholar, Dept. of Computer Science, S.V. University, Tirupati, India

P. Govindaraju**
Retd. Professor, Dept. of Computer Science, S.V. University, Tirupati, India

Abstract
Discovering bugs in software towards quality of software is given paramount importance in research areas. Towards this end automatic test case generation became essential as manual test data generation and adding test oracles is tedious task. It is more so when there are no formal specifications to unearth the faults in test outcome. Therefore, it is important to generate representative test sets that ensure complete code coverage. Genetic Algorithms are proved to be very useful for generation of unit tests and well suited for testing object oriented software systems. They are well known for their capabilities to test complex objects through sequences of method invocations. In this paper we used genetic algorithm for generating representative test suite for mutation testing. We built a tool that demonstrates the proof of concept. The empirical results are encouraging.

1. Introduction
It is a well known fact that software testing is an indispensable part of System Development Life Cycle (SDLC) which improves quality of software under test (SUT). Unit testing is widely used for unearthing bugs in SUT [18]. Each test contains test data required to execute specific program and the expected output. Traditionally many techniques came into existence over the years to generate test cases for discovering faults in SUT. Automatic test case generation is complex and challenging task [28]. In the same fashion automated test data generation is also difficult task[28]. Search based test data generation is explored in [14], [31]. Recently the focus was switched towards generation of test suites for high code coverage. Still there is a problem of determining the expected outcome which is also known as oracle problem. Therefore it is important to take care of automatic test suite generation and test oracles. For many years, as found in the literature, it was common practice to generate a test case for every coverage goal and combine all test cases to form test suite. A problem in this approach is that the size of test suite becomes unpredictable. The reason behind this is that test case generated for fulfilling one goal might be useful for other goals as well. This characteristic is exploited of late to produce representative test suite that not only covers the whole code besides ensuring smaller size test suite. There are many problems when one goal is targeted at a time. For instance, some branches are difficult to consider for coverage while some branches are infeasible. It is still open issues to find out how much time needs to be spent and difficulty prediction of coverage goals. In this paper we proposed a test suite generation approach using Genetic Algorithm (GA) for producing representative test suite generation that satisfies all coverage goals. The generated test suite can be used to have mutation testing so that the bugs in the SUT can identify the hidden bugs. Mutation testing is fault-based testing that is widely used [19],[26]. The case study example is in Fig. 1 and Fig. 2 which is used for empirical study. The example simulates a real world vending machine which takes coin as input and gives the requested.

Instead of generating test cases independently and then combining them into a test suite, our approach is to generate whole test suite that is representative of all
Author Profile

Prakasa Rao Chapani received Master of Computer Applications degree from Madras University and Master of Technology degree in Computer Science & Engineering from Acharya Nagarjuna University. He is a research scholar in the department of computer science, Sri Venkateswara University. His research focus is on Software Testing to improve the Quality under Software Project Management perspective.

Prof P.Govindarajulu, Professor at Sri Venkateswara University, Tirupathi, has completed M.Tech., from IIT Madras (Chennai). Ph.D from IIT Bombay (Mumbai). His area of research are Database, Data Mining, Image processing and Software Engineering.
coverage goals. All test cases in test suite are generated at a time and the fitness function used in GA takes care of testing goals simultaneously. The GA is used to optimize the generation of test suite. GA technique starts with initial population that contains test suite which is randomly generated.

\[
\begin{align*}
\text{StringBuffer} & \text{ ch} = \text{new StringBuffer("ax")}; \\
VendingMachine & \text{ v = new VendingMachine();} \\
v.\text{addChoc}(&"c1"); \\
v.\text{addChoc}(&"c2"); \\
v.\text{addChoc}(&"c3"); \\
v.\text{coin}(0); \\
v.\text{coin}(2); \\
v.\text{coin}(100); \\
\text{int ch} = v.\text{getChoc}(\text{cho}); \\
\text{System.out.println("First get, c. = choc -", ch, "; ch - ch");} \\
v.\text{choc}(); \\
ch = v.\text{getChoc}(\text{cho}); \\
\text{System.out.println("Second get, c. = choc -", ch, "; ch - ch");}
\end{align*}
\]

Fig. 2. Test suite containing multiple tests for VendingMachine class

The generated test suite represents the test coverage goals. Significantly smaller test suite is generated when compared with test suites generated based on a single goal in mind. Our solution towards generating whole test suite is representative of all coverage goals so as to reduce the size of test suite. Whole test suite generation was explored earlier in [18], [26], [27], [18], [34]. Our contributions in this paper are as follows:

- We developed a genetic algorithm that takes care of generation of a representative test suite generation at a time in order to have smaller size test suite besides ensuring full code coverage. The generated test suite can also be used for mutation testing.
- We built a tool that is designed to be modular and planned to be extended in our future research. It is meant for demonstrating the proof of concept with respect to test suite generation with minimal size and full coverage. Mutant killing test cases are directly generated from the given source code.

The remainder of the paper is structured as follows. Section 2 reviews literature on the prior works in test suite generation. Section 3 presents the proposed algorithm and methodology. Section 4 presents results of the experiments. Section 5 concludes the paper besides providing directions for future work.

2. Related Works


Malek and Fraser [31] proposed a hybrid approach using constraint and search-based testing to test software. Fraser and Zeller [28] focused on mutation-driven generation of test cases and test oracles. Arcuri and Fraser [14] also found it useful to have parameter tuning with respect to search-based software engineering. Many researchers contributed to test suite generation, mutation testing and automated testing of SUT as explored in [31]-[32].

3. Proposed Approach to Test Suite Generation

In this section we describe the search-based approach we followed towards generating test suite which is representative of testing goals and also maximizes mutation score. With respect to search-based testing, genetic algorithms have been around for many years for leveraging test data derivation. In fact they are most popular and they are of meta-heuristic in nature. The GA takes initial population which is randomly generated and reproduction is made iteratively using operators like crossover and mutation. Thus the GA is one of the evolutionary algorithms which make use of fitness function to have optimal solutions. This paper considers the generation of test suite for objects oriented programming. In fact, we tested the proposed solution with Java source code. A test case is considered a set of statements denoted as \( t = \{s_1, s_2, s_3, \ldots, s_l \} \) of length \( l \). In the conventional approach test suites are generated based on individual goals. In this paper,
we considered whole test suite generation that considers all goals at a time and the test suite gets generated with smaller size besides full coverage. Especially we used branch coverage approach that guides the test suite generation. Fitness function is used keeping branch coverage in mind which estimates the closeness of test suite with respect to all branches (if, while etc.) in the program.

Fig. 3 shows the general framework that is used for test suite generation and mutation testing. The framework starts with source code configuration. It takes Java source code (currently works for Java application testing) as input and starts the genetic algorithm steps. In the process it makes use of mutations iteratively and performs chromosome operations. JUnit Test Case is used in order to have test cases. It has assertions process as well. Finally the test cases are evaluated and some mutants are killed. The final test suite is presented. Mutation operators we applied on Java byte-code level.

Operators of various kinds used in Java program are used to have mutants generated. Each operator can have different ways of execution and all possible ways are considered to have mutations. The mutation operators are used to perform various operations. The operators include delete call, delete field, insert unary operator, replace arithmetic operator, replace bitwise operator, replace comparison operator, replace constant, and replace variable.

4. Experimental Results

We built a tool in Java programming language. It is menu-driven and intuitive in nature. It takes Java source code as input and performs mutation testing through whole test suite generation. For given Java class presented in Fig. 1 many operators are involved and different mutants are generated. The sample details are as presented in Table 1. Table 2, Table 3, Table 4 and Table 5. The details are

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>AORB B</td>
<td>AORB_1</td>
<td>credit = credit * coin; (line 30) void_coin(int); credit + coin =&gt; credit * coin</td>
<td></td>
</tr>
<tr>
<td>AORB B</td>
<td>AORB_2</td>
<td>credit = credit / coin; (line 30) void_coin(int); credit + coin =&gt; credit / coin</td>
<td></td>
</tr>
<tr>
<td>AORB B</td>
<td>AORB_3</td>
<td>credit = credit % coin; (line 30) void_coin(int); credit + coin =&gt; credit % coin</td>
<td></td>
</tr>
<tr>
<td>AORB B</td>
<td>AORB_4</td>
<td>credit = credit - coin; (line 30) void_coin(int); credit + coin =&gt; credit - coin</td>
<td></td>
</tr>
<tr>
<td>AORB B</td>
<td>AORB_5</td>
<td>change = credit * 90; int_getCho(jav; langu StrngBuff et) credits - 90 =&gt; credit * 90</td>
<td></td>
</tr>
<tr>
<td>AORB B</td>
<td>AORB_6</td>
<td>change = credit / 90; int_getCho(jav; lang StrngBuff et) credits - 90 =&gt; credit / 90</td>
<td></td>
</tr>
<tr>
<td>AORB B</td>
<td>AORB_7</td>
<td>change = credit % 90; int_getCho(jav; lang StrngBuff et) credits - 90 =&gt; credit % 90</td>
<td></td>
</tr>
<tr>
<td>AORB B</td>
<td>AORB_8</td>
<td>change = credit % 90; int_getCho(jav; lang StrngBuff et) credits - 90 =&gt; credit % 90</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>AORIS AORIS 1</td>
<td>if ((coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 2</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 3</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 4</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 5</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 6</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 7</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 8</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 9</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 10</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
<tr>
<td>AORIS AORIS 11</td>
<td>if (coin &lt;= 10 &amp; &amp; coin &gt;= 25)</td>
<td>void_coin(int); coin</td>
<td></td>
</tr>
</tbody>
</table>
The results revealed that the mutants generated through whole test suite generation, it was proved that the algorithm is capable of generating test suite which is representative of all goals and the branch coverage is given main focus. The mutation score is computed by the application which tells the coverage dynamics. The higher code coverage can be achievable by getting less mutation score.

Table 3 - Some of the Results of ROR operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR</td>
<td>ROR_1</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; coin &gt; 10</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; coin &gt; 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_2</td>
<td>if (coin &lt; 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25)</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; coin &lt; 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_3</td>
<td>if (coin &lt; 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25)</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; coin &lt; 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_4</td>
<td>if (coin &lt; 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25)</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; coin &lt; 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_5</td>
<td>if (coin &lt; 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25)</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; coin &lt; 10</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_6</td>
<td>if (true &amp; &amp; coin = 25 &amp; &amp; coin = 25)</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; true</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_7</td>
<td>if (false &amp; &amp; coin = 25 &amp; &amp; coin = 25)</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; false</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_8</td>
<td>if (coin &gt; 10 &amp; &amp; coin &gt; 25 &amp; &amp; coin &gt; 25)</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; coin &gt; 25</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_9</td>
<td>if (coin &gt; 10 &amp; &amp; coin &gt; 25 &amp; &amp; coin &gt; 25)</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; coin &gt; 25</td>
</tr>
<tr>
<td>ROR</td>
<td>ROR_10</td>
<td>if (coin &gt; 10 &amp; &amp; coin &gt; 25 &amp; &amp; coin &gt; 25)</td>
<td>(line 24) void_coin(int); coin = 10 =&gt; coin &gt; 25</td>
</tr>
</tbody>
</table>

Table 4 - Results of AOU operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOU</td>
<td>AOU_1</td>
<td>credit += coin;</td>
<td>(line 30) void_coin(int).credit += -credit</td>
</tr>
<tr>
<td>AOU</td>
<td>AOU_2</td>
<td>return - change</td>
<td>(line 40) int_getChoice(StringBuffer).change += -change</td>
</tr>
<tr>
<td>AOU</td>
<td>AOU_3</td>
<td>change = -credit + 90</td>
<td>(line 42) int_getChoice(StringBuffer).credit += -credit</td>
</tr>
<tr>
<td>AOU</td>
<td>AOU_4</td>
<td>return - change</td>
<td>(line 45) int_getChoice(StringBuffer).change += -change</td>
</tr>
<tr>
<td>AOU</td>
<td>AOU_5</td>
<td>return - credit</td>
<td>(line 77) int_getChoice(StringBuffer).credit += -credit</td>
</tr>
</tbody>
</table>

Table 5 - Results of COR operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutant Name</th>
<th>Mutant Content</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td>COR</td>
<td>COR_3</td>
<td>if (coin &gt; 10)</td>
<td>(line 24) void_coin(int); coin = 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25</td>
</tr>
<tr>
<td>COR</td>
<td>COR_2</td>
<td>if (coin &gt; 10)</td>
<td>(line 24) void_coin(int); coin = 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25</td>
</tr>
<tr>
<td>COR</td>
<td>COR_3</td>
<td>if (coin &gt; 10)</td>
<td>(line 24) void_coin(int); coin = 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25</td>
</tr>
<tr>
<td>COR</td>
<td>COR_4</td>
<td>if (coin &gt; 10)</td>
<td>(line 24) void_coin(int); coin = 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25</td>
</tr>
<tr>
<td>COR</td>
<td>COR_5</td>
<td>if (coin &gt; 10)</td>
<td>(line 24) void_coin(int); coin = 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25</td>
</tr>
<tr>
<td>COR</td>
<td>COR_6</td>
<td>if (coin &gt; 10)</td>
<td>(line 24) void_coin(int); coin = 10 &amp; &amp; coin = 25 &amp; &amp; coin = 25</td>
</tr>
</tbody>
</table>

As can be seen in Figure 4, it is evident that the applications are presented in terms of the number of lines of code.
As can be seen in Figure 5, it is evident that there is performance difference in generating test suites based on the size of the SUT.

As can be seen in Figure 6, it is evident that the number of mutants differs from each application.

As can be seen in Figure 7, it is evident that the results reveal the relationships between average branch coverage and average mutation score (proposed) using scatter chart.

5. Conclusions and Future Work

In this paper we studied the concept of generation of representative test suite which ensures complete code coverage. Coverage criteria play an important role in automatic test case generation. Traditional approaches targeted one particular coverage goal. From the recent experiments in software testing it is evident that the optimization of whole test suite generation is far better than the traditional method of targeting one coverage goal. Genetic algorithms have been applied successfully to generate unit tests for testing object oriented software. GA is one of the search based algorithms widely used to generate test cases. In this paper we applied GA for generating representative test suites and mutation testing. We built a tool to demonstrate the proof of concept. Mutation testing became easy with generation of test suite that covers all goals. The empirical results revealed that the application is capable of generating whole test suite which is representative of all test goals besides keeping it small in size. Our future work is to improve the tool for test suite prioritization.

Acknowledgments

I take this opportunity to sincerely acknowledge Dr. Kancharla Ramaiah, correspondent of Prakasam Engineering College for providing all the facilities, which buttressed me to perform my work comfortably. Foremost, I would like to express my sincere gratitude to fellow members of the teaching staff at the Prakasam Engineering College. My sincere thanks also goes to my Uncle Dr. C. Subba Rao for his inspiration and sparing his precious time. Last but not the least, I would like to thank my family: my parents Murali Krishna Rao and
Vasanthi Lakshmi, for giving birth to me at the first place and supporting me spiritually throughout my life.

References

[31] Vincenzo Humberto Serapiglia Durcella, Rodrigo Frasino Araujoa, b and Marco Aurélio Garejato Silva. (2013). A scoping study on the 25 years of research into software testing in Brazil and an outlook on the future of the area. p935-950.

*
Prakasa Rao Chapati received Master of Computer Applications degree from Madras University and Master of Technology degree in Computer Science & Engineering from Acharya Nagarjuna University. He is a research scholar in the department of computer science, Sri Venkateswara University. His research focus is on Software Testing to improve the quality under Software Project Management perspective.

Prof. P. Govindaraju, Professor at Sri Venkateswara University, Tirupathi, has completed M.Tech., from IIT Madras (Chennai), Ph.D from IIT Bombay (Mumbai). His area of research are databases, Data Mining, Image processing and Software Engineering.
Automatic Discovery of Dependency Structures for Test Case Prioritization

C. Prakasa Rao
Research Scholar, Dept. of Computer Science, S.V. University, Tirupati, India.

P. Govindarajulu
Retd. Professor, Tirupati, India Dept. of Computer Science, S.V. University.

Abstract
In software engineering "testing" is one of the phases in system development life cycle. Functional test suites are used to discover bugs in Software Under Test (SUT) and improve its quality. A good test suite uncovers more faults in the SUT. As test suite contains many test cases, the order of their execution plays an important role in increasing the rate of fault detection which can provide early feedback to development team so as to help them to improve the quality of the software. Therefore it is very useful to prioritize test cases that will lead to the increase in the rate of fault detection. However, prioritization of functional test suites is a challenging problem to be addressed. Recently Haidry and Miller proposed a family of test case prioritization techniques that use the dependency information from a test suite to prioritize that test suite. The nature of the techniques preserves the dependencies in the test ordering. Dependencies in test cases can have their impact on the discovery of faults in software. This hypothesis has been proved by these authors as their empirical results revealed it. However, they do not automate the extraction of dependency structures among the test cases that can help in effective prioritization of functional test suites. In this paper we propose a methodology that automates the process of extraction of dependency structures from the test cases that will result in the increase the rate of fault detection. Thus the number of bugs uncovered from the software under test is improved. This leads to the improvement of quality of the software.

Index Terms
Software engineering, testing, test case prioritization, dependency structures

1. Introduction
Test suites can help detect faults in SUT. Provided this goal is achieved, there are many issues with it. For instance test suites when executed in particular sequence can provide chances to unearth more faults. It does mean that test suite prioritization can be used to optimize testing results or to uncover more hidden faults. In order to achieve this, it is possible to find dependency structure that can be used to priorities test suites. The functional test suites when subjected to prioritization can give effective test results that can help developers to rectify problems in SUT. Haidry and Miller [1] focused on the process of test suite prioritization. They used a hypothesis “dependencies among test cases can have their influence on the rate of fault detection”. Thus the test case prioritization is given importance. It is a process of ensuring that the test cases are executed in proper sequence in order to achieve high rate of fault detection. The rate of fault detection is measured using the number of faults detected. As some tests should occur before other tests, it is ensured to prioritize test cases so as to achieve optimal results [1]. Sample dependency structure can be visualized in Figure 1.

Fig. 1 - Sample dependency structure

As seen in Figure 1, it is evident that the root nodes are independent of other nodes and they do not have dependencies. Dependencies are of two types namely direct and indirect. For example in Fig.2 D6 is a direct dependent of D3 and indirectly dependent on 11. Yet in another classification, dependency is of two types namely open dependency and closed dependency. Open dependency is the fact that a test case is executed before another one but need not be necessarily just immediately before the test case. The closed dependency is opposite to it where a test case needs to be executed immediately before the other test case. The combination of open and closed dependencies is also possible for optimal results. To measure dependencies, two measures are used. They are known as DSP height and DSP volume. Dependency Structure Prioritization volume refers to the count of dependencies while the DSP height indicates the depth in dependency levels. Direct and indirect dependencies are considered while computing DSP volume. On the other hand, the height of all test paths is considered for computing DSP height. The two graph measures are used for best ordering of test cases for optimal results. Experiments are made with open dependencies and closed dependencies. Many real time projects were considered for experiments. Out of them Dash is recorded to have highest dependencies and CRM1 and CRM2 recorded the lowest. Many SUTs were tested with prioritization of test cases.
The experiments are useful to know the fault detection rate when dependency structures are used for prioritization. A measure used in [6] is known as Average Percentage of Faults Detected (APFD) for fault detection. The more APFD value is the more in the rate of detection of faults in SUT. All SUTs are tested with APFD measures under open and closed dependencies. Many DSP prioritization methods were considered and some other methods that do not use DSP measures can also be used in the process. The experimental results showed that DSP prioritization methods achieved higher results while non-DSP prioritization methods could not achieve high rate of fault detection. The empirical results proved the fact that DSP measures were able to increase the rate of fault detection for any given SUT. As explored in [1] there are many test case prioritization techniques. They are model – based [12], history – based [11] and knowledge-based [2]. The first one uses model of the system, second one uses past execution cycles, and third one uses human know how of the task for the purpose of test case prioritization. Our contributions in this paper are as follows.

We proposed a methodology for automatic discovery of dependency structures from SUT. This methodology guides the program to obtain dependency structures and helps in prioritization of test cases.

We proposed an algorithm that makes use of discovered dependency structures and prioritizes test cases automatically.

We evaluate the functions such as automatic discovery of dependency structures and also the test case prioritization with empirical study using the tool built to demonstrate the proof of concept.

The remainder of the paper is structured as follows. Section II reviews literature on the prior works. Section III presents the proposed methodology for automatic discovery of dependency structures and algorithm for prioritizing test cases. Section IV presents evaluation of the proposed work while Section V provides conclusions and recommendations for future work.

2. Related Works

This section provides review of literature on prior works. In 1997 Wong et al. [1] proposed a hybrid approach for regression testing which uses the combination of approaches like minimization, modification, and prioritization-based selection. The purpose of regression testing is to ensure that changes made to software, such as adding new features or modifying existing features, have not adversely affected features of the software that should not change. Regression testing is usually performed by running some, or all, of the test cases created to test modifications in previous versions of the software. Many techniques have been reported on how to select regression tests so that the number of test cases does not grow too large as the software evolves. Our proposed hybrid technique combines modification, minimization and prioritization-based selection using a list of source code changes and the execution traces from test cases run on previous versions. This technique seeks to identify a representative subset of all test cases that may result in different output behavior on the new software version [1].

Ryu and Glinz [6] discussed about scenarios or use cases that can be used to capture requirements. The modeling tools such as UML also do not have scenario based dependencies. They opined that verification and validation are important activities in software development process. It is true in the case of test case generation and execution as well. They proposed a new model to find dependencies between scenarios. In this paper we focused on the dependencies among methods while [6] explored dependencies among the scenarios. Dependency charts were built in order to help test engineers to test the SUT in systematic fashion so as to discover more bugs. Elbaum et al. [11] focused on test case prioritization by considering fault severities and varying test costs. The regression testing can take the help of prioritization results in order to improve the possibilities of finding and fixing bugs. APFD measure is used to know the rate of fault detection. The previous uses of APFD were made when severities and test costs are uniform. In [11] a new technique is proposed in order to assess the rate of fault detection with prioritized test cases. Thus priority based reuse of test suits save more time to software engineers besides helping them in discovering more bugs. The new technique was an improved form of APFD that is based on test costs and the severities.

Rothermel et al. [2] focused on cost-effectiveness of regression testing with respect to test suite granularity. Since regression testing is an expensive test process, the cost can be reduced with the methods that are cost-effective. Towards this prioritization of test cases play an important role in order to make it less costly but still being able to discover more bugs. The bottom line of the research is to reduce the cost and also increase the rate of fault detection. Elbaum et al. [12] made an empirical study on test case prioritization. The aim of their research is to reduce the cost of regression testing. The end result expected is the same “increasing the rate of fault detection”. One potential goal of test case prioritization is that of increasing a test suite’s rate of fault detection—a measure of how quickly a test suite detects faults during the testing process. An improved rate of fault detection can provide earlier feedback on the system under test, enable earlier debugging, and increase the likelihood that, if testing is prematurely halted, those test cases that offer the greatest fault detection ability in the available testing time will have been executed [12].
Peirce’s criterion is a rigorous method based on probability theory that can be used to eliminate data 
“outliers” or spurious data in a rational way. Currently, another method called Chauvenet’s criterion is used in 
many educational institutions and laboratories to perform this function. Although Chauvenet’s criterion is well 
established, it makes an arbitrary assumption concerning the rejection of the data. Peirce’s criterion does not make 
this arbitrary assumption. In addition, Chauvenet’s criterion makes no distinction between the case of one or 
several suspicious data values whereas Peirce’s criterion is a rigorous theory that can be easily applied in the case of 
several suspicious data values. In this paper, an example is given showing that Peirce’s and Chauvenet’s criterion give 
different results for the particular set of data presented.[13]

Code prioritization for testing promises to achieve the maximum testing coverage with the least cost. This paper 
presents an innovative method to provide hints on which part of code should be tested first to achieve best code 
coverage. This method claims two major contributions. First it takes into account a “global view” of the execution 
of a program being tested, by considering the impact of calling relationship among methods/functions of complex 
software. It then relaxes the “guaranteed” condition of traditional dominator analysis to be “at least” relationship 
among dominating nodes, which makes dominator calculation much simpler without losing its accuracy. It 
also then expands this modified dominator analysis to include global impact of code coverage, i.e. the coverage 
of the entire software other than just the current function. We implemented two versions of code prioritization 
methods, one based on original dominator analysis and the other on relaxed dominator analysis with global view.[4].

Software engineers often save the test suites they develop so that they can reuse those test suites later as their 
software evolves. Such test suite reuse, in the form of regression testing, is pervasive in the software industry. 
Running all of the test cases in a test suite, however, can require a large amount of effort; for example, one of our 
industrial collaborators reports that for one of its products of about 20,000 lines of code, the entire test suite requires 
seven weeks to run. In such cases, testers may want to order their test cases so that those with the highest priority, 
according to some criterion, are run earlier than those with lower priority.[10].

Test case prioritization techniques have been shown to be beneficial for improving regression-testing activities. With 
prioritization, the rate of fault detection is improved, thus allowing testers to detect faults earlier in the system-
testing phase. Most of the prioritization techniques to date have been code coverage-based. These techniques may 
treat all faults equally. We build upon prior test case prioritization research with two main goals: (1) to improve 
user perceived software quality in a cost effective way by considering potential defect severity and (2) to improve 
the rate of detection of severe faults during system level testing of new code and regression testing of existing code. 
We present a value-driven approach to system-level test case prioritization called the Prioritization of 
Requirements for Test (PORT). PORT prioritizes system test cases based upon four factors: requirements volatility, 
customer priority, implementation complexity, and fault proneness of the requirements. We conducted a PORT 
case study on four projects developed by students in advanced graduate software testing class. Our results show 
that PORT prioritization at the system level improves the rate of detection of severe faults. Additionally, customer 
priority was shown to be one of the most important prioritization factors contributing to the improved rate of 
fault detection [3].

Test engineers often possess relevant knowledge about the relative priority of the test cases. However, this knowledge 
can be hardly expressed in the form of a global ranking or scoring. In this paper, we propose a test case prioritization 
technique that takes advantage of user knowledge through a machine learning algorithm, Case-Based Ranking (CBR). 
CBR elicits just relative priority information from the user, in the form of pair wise test case comparisons. User input 
is integrated with multiple prioritization indexes, in an iterative process that successively refines the test case 
ordering. Preliminary results on a case study indicate that CBR overcomes previous approaches and, for moderate 
suite size, gets very close to the optimal solution [7].

Regression testing is an expensive part of the software maintenance process. Effective regression testing 
techniques select and order (or prioritize) test cases between successive releases of a program. However, 
selection and prioritization are dependent on the quality of the initial test suite. An effective and cost efficient test 
generation technique is combinatorial interaction testing, CIT, which systematically samples all t-way combinations 
of input parameters. Research on CIT, to date, has focused on single version software systems. There has been little 
work that empirically assesses the use of CIT test generation as the basis for selection or prioritization. In this paper 
we examine the effectiveness of CIT across multiple versions of two software subjects. Our results show that CIT 
performs well in finding seeded faults when compared with an exhaustive test set. We examine several 
CIT prioritization techniques and compare them with a re-
generation/prioritization technique [14].

Test case prioritization techniques have been empirically 
proved to be effective in improving the rate of fault 
detection in regression testing. However, most of previous 
techniques assume that all the faults have equal severity, 
which does not meet the practice. In addition, because 
most of the existing techniques rely on the information 
gained from previous execution of test cases or source 
code changes, few of them can be directly applied to non-
regression testing. In this paper, aiming to improve the rate of severe faults detection for both regression testing and non-regression testing, we propose a novel test case prioritization approach based on the analysis of program structure. The key idea of our approach is the evaluation of testing-importance for each module (e.g., method) covered by test cases. As a proof of concept, we implement a prioritization tool, and perform an empirical study on two real, non-trivial Java programs. The experimental result represents that our approach could be a promising solution to improve the rate of severe faults detection.

Regression testing assures changed programs against unintended amendments. Rearranging the execution order of test cases is a key idea to improve their effectiveness. Pardoxically, many test case prioritization techniques resolve tie cases using the random selection approach, and yet random ordering of test cases has been considered as ineffective. Existing unit testing research unveils that adaptive random testing (ART) is a promising candidate that may replace random testing (RT). In this paper, we not only propose a new family of coverage-based ART techniques, but also show empirically that they are statistically superior to the RT-based technique, in detecting faults. Pair-wise comparison has been successfully utilized in order to prioritize test cases by exploiting the rich, valuable and unique knowledge of the tester. However, the prohibitively large cost of the pair wise comparison method prevents it from being applied to large test suites. In this paper, we introduce a cluster-based test case prioritization technique. By clustering test cases, based on their dynamic runtime behavior, we can reduce the required number of pair-wise comparisons significantly. The approach is evaluated on seven test suites ranging in size from 154 to 1,061 test cases. We present an empirical study that shows that the resulting prioritization is more effective than the existing coverage-based prioritization techniques in terms of rate of fault detection.

3. Methodology For Automatic Discovery Of Dependency Structures

The research on test case prioritization focused on various approaches as found in the previous section. For instance, they are based on execution traces [1], dependency charts that are derived through scenario-based testing [6], test costs and fault severities [11], test suite granularity and its impact on cost-effectiveness on regression testing [2], comparator techniques, statement level techniques and function level techniques [12], cost prioritization [4], fine granularity and coarse granularity [9]. Prioritization of Requirements for Test (PORT) which is a value-driven approach [3], use case based ranking methodology [7], combinatorial interaction testing [14], analysis of program structure [15], adaptive random test case prioritization [5] and clustering test cases [9]. More recently Haidry and Miller [15] used dependency structures for test case prioritization. In this paper, we improve the approach used in [15] by discovering dependency structures automatically. The architecture of the proposed methodology is as shown in Figure 2.

Figure 2: Proposed methodology

As can be seen in Figure 2, it is evident that the proposed methodology depends on program execution traces and the actual program. The method discovery process makes a list of all methods available and in fact the methods are discovered using reflection API. The call processing component is responsible to use traces and have some meta data associated with calls. This meta data is used later for test case prioritization. The test case prioritization component is responsible to understand the meta data associated with all calls and also considers test suite. As can be seen in the Figure 2, the prioritization technique is divided into two phases: a data phase where dependency structures are extracted and a prioritization phase where test cases are prioritized.

TCP (Test Case Prioritization) Algorithm

Input : Execution traces (ET) and program (P). Test Suite (TS)
Output: Prioritized test cases (PT)
1. Initialize a vector (M)
2. Initialize another vector (MM)
3. Discover methods from P and populate M
4. for each method m in M
   a. scan TS
   b. associate meta data with calls
   c. add method m to vector MM
5. end for
6. for each mm in MM
   a. analyze TS
   b. correlate with mm
   c. add corresponding m to PT
7. return PT

Algorithm for test case prioritization
As can be seen in listing 1, it is evident that the proposed method takes traces, program and test suite as input. It performs discovery of methods and automatic discovery of dependencies in the form of methods associated with meta data and finally performs prioritization of test cases in the given test suite.

4. Experimental Results

The tool implemented in our previous work has been extended to incorporate the functionality of the proposed methodology in this paper. The tool demonstrates the proof of concept and discovers dependency structures from given program. The tool can distinguish between open and closed dependencies as described earlier in this paper. The inputs and outputs are presented in this section besides the results of experiments. Open dependency related input program is as shown in Listing 2.

As can be seen in Figure 3, it is evident that the closed dependencies are presented graphically. The closed dependencies as per the given input file are shown. The application can work for any input file so as to discover closed dependencies.

As can be seen in Figure 4, it is evident that the open dependencies are presented graphically. The open dependencies as per the given input file are shown. The application can work for any input file so as to discover open dependencies.

As can be seen in Figure 5, it is evident that the open and closed dependencies are presented graphically. The dependencies as per the given input file are shown. The application can work for any input file so as to discover open dependencies. The source code of these dependencies is found in appendix.

5. Evaluation

For evaluating our work specific procedure is followed as described here. First, the discovery of dependencies is done manually by human experts. The input file is shared with expert software engineers who have testing knowhow. The human experts studied the given inputs and provided their results which are done manually. Their results are saved and they reflect the ground truth. Later on our application is tested with same inputs. This process is continued for many Java applications to be tested. The results of manual discovery of dependencies (closed and open) are compared with the results discovered by our application. Around 100 times this evaluation of the application results by comparing with ground truth consistently resulted in the same. Thus 100% accuracy has been recorded by the application. When time is compared, human experts took 10 to 15 minutes to discovery dependencies in average while our application takes negligible time to show the dependencies.
Many experiments proved that the automatic discovery of dependency structures do match with the ground truth and the tool has been extended to prioritize test cases automatically. In our previous paper we focused on test suite generation while this paper while this paper focused on automatic discovery of dependency structures for test case prioritization. More details on our tool will be presented in our next paper.

6. Conclusions and Future Work

Test case prioritization has its utility in improving the rate of fault detection in SUT. As test suite contains many test cases, the order of their execution plays an important role in increasing the rate of fault detection which can provide early feedback to development team so as to help them to improve the quality of the software. Therefore it is very useful to prioritize test cases that will lead to the increase in the rate of fault detection. In this paper we proposed a novel mechanism to discover dependency structures from SUT automatically and use them for prioritization of test cases. This work is very closer to that of Haidry and Miller. However, they did not automate the discovery of dependency structures. Dependencies are of two types namely direct and indirect. Both types are considered in this paper. We built a prototype application that demonstrates the proof of concept. The empirical results reveal that the automatic discovery of dependency structures can help in complete automation of test case prioritization. In future we integrate the whole test suite generation and test suite prioritization into a single tool that will help software engineering domain for automatic test case generation and test case prioritization.

REFERENCES


Prakasa Rao Chapati received Master of Computer Applications degree from Madras University and Master of Technology degree in Computer Science & Engineering from Acharya Nagarjuna University. He is a research scholar in the department of Computer Science, Sri Venkateswara University. His research focus is on Software Testing to improve the quality under Software Project Management perspective.

P.Govindaraju, Professor at Sri Venkateswara University, Tirupathi, has completed M.Tech., from '1' Madras (Chennai),Ph.D from IIT Bombay (Mumbai). His area of research are Databases, Data Mining, Image processing, and Software Engineering
Abstract

Testing has been an essential part of software development life cycle. Automatic test case and test data generation has attracted many researchers in the recent past. Test suite generation is the concept given importance which considers multiple objectives in mind and ensures core coverage. The test cases thus generated can have dependencies such as open dependencies and closed dependencies. When there are dependencies, it is obvious that the order of execution of test cases can have impact on the percentage of flaws detected in the software under test. Therefore test case prioritization is another important research area that complements automatic test suite generation in objects oriented systems. Prior researches on test case prioritization focused on dependency structures. However, in this paper, we automate the extraction of dependency structures. We proposed a methodology that takes care of automatic test suite generation and test case prioritization for effective testing of object oriented software. We built a tool to demonstrate the proof of concept. The empirical study with 20 case studies revealed that the proposed tool and underlying methods can have significant impact on the software industry and associated clientele.

Keywords: Software Engineering, Testing, Test Case Prioritization, Dependency Structures, Branch Coverage.

1. INTRODUCTION

Software testing can improve quality of Software Under Test (SUT). Unit testing [3] is one of the important tests made. However, generating test cases automatically has been around in the research circles. Nevertheless, it is a complex and challenging task [5]. Other approaches to test case generation are based on search [2], [4]. The recent focus is on the high coverage while generating test cases automatically. To solve this problem automatic test suite generation came into existence. Test suite can reduce number of test cases as it can provide representative test cases with high coverage. Test suite generation targets many goals at a time and generates optimal test suites. A problem with considering a single goal is that it cannot guarantee of high coverage and may produce infeasible ones. In this paper we proposed a mechanism for automatic test suite generation based on Genetic Algorithm (GA). The test suites thus generated are used for mutation testing which is widely used technique [2], [4]. GA is used to have an evolutionary approach to generate test suites that can best represent high code coverage.

Test suites are used to detect faults. However, there are some test cases that depend on others. This will provide way to further research on the dependencies and improving detection of fault
rates. In order to unearth hidden faults, it is essential to consider dependencies. To achieve this dependency structures are to be discovered. Haidry and Miller [8] focused on the test case prioritization but they did not discover dependency structures automatically. In this paper, we focus on the automatic discovery of dependencies. Figure 1 shows both open and closed dependencies.

As can be seen in Figure 1, the root nodes are I1 and I2 that are independent on other nodes. Other nodes have dependencies. Dependencies are classified into two categories. They are open and closed dependencies. The D6 is indirectly dependent on I1 and directly dependent on D3. Therefore there is closed dependency on D3 and open dependency on I1 (no immediate dependency). When both kinds of dependencies are considered, the quality of test case prioritization improves. DSP volume and DSP height are the two measures used in test case prioritization. DSP height indicates depth while DSP volume indicates the count of dependencies. In [6] different measure is used known as Average Percentage of Faults Detected (APFD). APFD value is directly proportional to fault detection rate.

In the literature many kinds of test case prioritization techniques were found. They are knowledge based [18], history based [11] and model based [25]. More on these techniques is covered in Related Works section of this paper. Our main contribution in this paper is the development of a comprehensive tool that supports test suite generation, automatic discovery of dependency structures and test case prioritization. The tool employs the methods we proposed in our earlier work to demonstrate the proof concept. The remainder of the paper is structured as follows. Section 2 reviews literature related to test case generation and test case prioritization. Section 3 focuses on comprehensive testing tool for automatic test suite generation and prioritization. Section 4 presents experimental results while section 7 concludes the paper besides providing future work.

2. RELATED WORKS
This section provides review of literature pertaining to whole test suite generation, automatic extraction of dependency structures and prioritization of test cases for improving bug detection rate.

2.1 Whole Test Suite Generation
Many researchers tried to generate test suites that can reduce number of test cases as they provide representative test cases. Moreover test suits can also provide maximum coverage. Towards achieving this, Rodolph [13] focused on genetic algorithms while Arcuri and Briand [26] threw light into adapting random testing. Bacteriologic algorithm [27], hybrid approach with static and dynamic methods [28], Directed Automated Random Testing (DART) [29] are the other approaches found in the literature. PathFinder [30], evolutionary programming [21], integrated evolutionary programming [31], Mock classes and test cases [32], exploring length of test cases...
for code coverage [33], JCrasher [34] are other approaches found for testing Java applications. Malburg and Fraser [2] combined both search-based and constraint based approaches to test software. Mutation-driven generation [1] and parameter tuning [3] and mutation testing [4], [7] were other useful researches found.

2.2 Automatic Generation of Dependency Structures

Ryser and Glinz [9] focused on dependency charts for reflecting dependencies among various scenarios for improving testing. They used natural language processing techniques for mapping system requirements and descriptions. State charts are used to refine the representations. This will have clear picture of dependency structures based on which test cases are extracted while making use of dependency charts. With respect to dependencies Kim and Bae [10] opine that there are two features namely accidental and essential in a system. The former depends on the latter thus producing different levels in the system. These researchers employed an approach to align the features pertain to the two classes of features based on the dependencies. The accidental features were found to be dynamic and changed often while the essential features do not change frequently. However, they only focused on the dependency structures and did not consider test case prioritization.

2.3 Test Case Prioritization

This section provides test case prioritization categories that are close to our approach in this paper. They are history-based, knowledge-based and model-based approaches.

2.3.1 History-Based Prioritization

Rothermel et al. [11] used execution information for improving rate of fault detection. Their techniques are based on code coverage, probability of fault finding, untested code and so on. A greedy algorithm is used to know tests that have highest coverage. Li et al. [12] focused on many non-greedy algorithms for test case prioritization. These algorithms include genetic algorithms, hill climbing, and so on. They achieved high fault detection rate. Jeffrey and Gupta [14] also depended on the paths covered while running the code. They found the affecting paths and considered them as relevant. The more relevancy demands more importance. Wong et al. [15] focused on the changes in the source code from the previous runs in order to complete test case prioritization with respect to regression testing. Li [16] followed an approach known as code-coverage-based technique for test case prioritization. He could find the relationships among code blocks and prioritize test cases based on the code coverage concept. He also coupled the solution with symbolic execution for more accuracy. Later on in [17] the technique was improved further which throws light into test suite generation and also priority in the execution of test cases.

2.3.2 Knowledge-Based Prioritization

Ma and Zhao [18] focused on program structure analysis in order to find faults in the software. They used ranking given to modules that indicated testing importance of the module. This is based on the importance of module and fault proneness. Their approach improved fault detection rate when compared with random prioritization. Krishna moorthi and Sahaaya [19] used software requirements as basis for test case prioritization. They used fault impact, traceability, completeness, complexity, implementation, and requirements change and customer priority for automatic test case prioritization. In [20] research was done in the similar lines. In [21] case-based ranking approach was followed. It is a machine learning based mechanism employed for test case prioritization. It depends on the user knowledge pertaining to program. Zhang et al. [28] used data mining approach known as clustering for test case prioritization. They rank the test cases based on the clustering results. In [22] combinatorial interaction testing is employed to define prioritization. Based on the interactions and benefit analysis test cases are prioritized. In [23], UML models are used to generate test case priorities based on the underlying interaction levels. Adaptive Random Testing [24] focused on utilization of knowledge that has been categorized to prioritize test cases. It was proved to be inexpensive when compared with other techniques.
2.3.3 Model-Based Prioritization

Kundu et al. [25] explored test case generation and then prioritization for object oriented programs. Their approach is to take UML sequence diagrams as input and generate sequence graphs and prioritize the test cases. It makes use of weights concept such as message weight and edge weight. Based on the weights certain metrics are used. They include sum of message weights of a path, sum of message weights for all edges and weighted average edge weights. The results reveal the interactions between the components of the system. This will also provide a kind of trace that can be used to prioritize test cases. In this paper we proposed mechanisms for whole test suite generation and test case prioritization besides testing them using a tool that we built in Java programming language.

3. COMPREHENSIVE TESTING TOOL FOR AUTOMATIC TEST SUITE GENERATION, PRIORITIZATION

We built a tool that facilitates automatic testing of object oriented applications. The tool includes features such as automatic test suite generation, automatic discovery of dependency structures and prioritization of test cases. The following sub sections provide insights of the underlying features of the tool. This work of ours is an extension to our previous work [45], [46], and [47].

3.1 Automatic Test Suite Generation

This section provides details of our approach for automatic test suite generation. It is a search-based approach to generate whole test suite based on testing goals. It also maximizes mutation score. For test data derivation for search based testing, GAs are widely used. In fact, GAs are meta-heuristic in nature and are very popular for search based testing. GA makes use of initial population and uses operators like mutation and crossover to have next generation. Thus GA is considered to be an evolutionary algorithm that can provide optimal solutions. In this paper, we consider the generation of test suite for object oriented source code developed in Java. Let t be the test case t = (s1, s2, s3, ..., sL) with certain length L. In the traditional approaches a single goal is used to generate test suites. However, in this paper, we use multiple goals at a time and generate whole test suite that is representative for all tests with high coverage. Branch coverage is used to guide test suite generation process. To this effect, fitness function is used. The methodology is shown in Figure 3.

![Diagram](image-url)

**FIGURE 2**: Methodology for proposed whole test suite generation.
As shown in Figure 2, the proposed solution starts with source code configuration. The framework takes Java source code and generates mutations in iterative fashion. It also involves in chromosome operations. JUnit is used for API required to generate test cases. JUnit has its assertions process. The generated test cases are evaluated and some of the mutants are eliminated or killed. The mutation operators are used at Java byte code level. The final test suite is generated. The GA operators are used for performing various operations. The operators at Java code level are used to have different mutants generated.

3.2 Automatic Discovery of Dependency Structures and Test Case Prioritization

There are many approaches found in the literature for test case prioritization. They include adaptive random test case prioritization [39], program structure analysis [44], combinatorial interaction testing [43], use case based ranked method [41], Prioritization of Requirements for Test (PORT) [37], cost prioritization [38], function level and statement level techniques [42], regression testing [36], test costs and severity of faults [42], and scenario-based testing [40]. The approach we followed in this paper is close to that of [8] with difference in the approach for test case prioritization. Overview of our architecture for test case prioritization is shown in Figure 3.

As shown in Figure 2, program execution traces and program given as input. The method discovery is the process of finding methods in the program using reflection API. The list of methods is further used to prioritize test cases. The call processing is the process of identifying the method calls in the traces and determining the order in which methods are to be tested in order to have high level of fault detection rate. The meta data associated with calls can help in making well informed decisions. The test case prioritization module is responsible to actually prioritize test cases that are supplied to it in the form of test suite and provide results. The results contain test cases that have been prioritized. The priority when used will improve the percentage of fault detection.

3.3 Test Case Prioritization Algorithm

This sub section provides the pseudo code for the test case prioritization. This code is implemented in our tool in order to demonstrate the proof of concept.
Pseudo Code for Test Case Prioritization

Input : Execution Traces(ET), Program(P) and Test Suite(TS)
Output : Prioritized Test cases(PT)

Step 1 : Initialize a vector (M)
Step 2 : Initialize another vector (MM)
Step 3 : Discover methods from P and populate M
Step 4 : for each method m in M
     step 4.1 : scan TS
     step 4.2 : associate meta data with calls
     step 4.3 : add method m to vector MM
Step 5 : end for
Step 6 : for each mm in MM
     step 6.1 : analyze TS
     step 6.2 : correlate with mm
     step 6.3 : add corresponding m to PT
Step 7 : end for
Step 8 : return PT

As seen, the Pseudo Code for implementing test-case prioritization. The algorithm takes execution traces, program and test suite. After processing, it generates prioritised test cases. First of all it discovers methods and makes a collection object to hold it. For each method test suite is verified and the meta data is associated with calls. This is an iterative process which ends with a vector or collection containing methods associated. Then the test case prioritization considers the dependencies and test suites available to have a list of test cases in the order of priority. Prioritized test cases are the final result of the algorithm.

4. EXPERIMENTAL RESULTS
The tool we built was tested with 20 real time applications. The details of the applications in terms of number of classes, number of branches and LOC are as shown in Figure 4. The tool demonstrates the proof of concept and discovers dependency structures from given program. The tool can distinguish between open and closed dependencies as described earlier in this paper. The tool supports both automatic test suite generation and test case prioritization with automated discovery of dependency structures.
As can be seen in Figure 4 the case studies considered for experiments include Colt, Commons CLI, Commons Codec, Commons Collections, Commons Math, Commons Primitives, Google Collections, Industrial Case Study, Java Collections, JDom, JGraphT, Joda Time, NanoXML, Numerical Case Study, Java Regular Expressions, String Case Study, GNU Trove, Xmlenc, XML Object Model and Java ZIP Utilis. The experiments are made with proposed approach and the similar approach EvoSuite besides a single objective approach.
As shown in Figure 5, it is evident that the proposed approach is compared with other approaches. The proposed approach is better than the single objective approach. This is because generating test suite with multiple objectives in mind can reduce number of test cases besides being able to cover all branches. With respect to average branch coverage Evosuite and the proposed method outperform single branch approach.

As shown in Figure 6, the average branch coverage of the proposed system and Evosuite perform better than the single branch strategy. The reason behind this is that when test suite is generating with multiple objectives in mind, the generated test cases will be less besides being able to cover all possible branches. In horizontal axis case studies and in vertical axis the average branch coverage is presented.
As shown in Figure 7, the average test suite length of the proposed system and Evosuite perform better than the single branch strategy. The reason behind this is that when test suite is generated with multiple objectives in mind, the generated test cases will be less besides being able to cover all possible branches. In horizontal axis case studies and in vertical axis the average length is presented.

As shown in Figure 8, the case study applications and their statistics in terms of lines of code, number of functions and number of dependencies are presented. These case studies are used to apply the proposed test case prioritization approach with automated discovery of dependencies.
The results revealed that the prioritization can help increase the number of faults detected. This is the ultimate aim of the proposed tool in this paper.

As shown in Figure 9, it is evident that the percentage of test suites executed is presented in horizontal axis while the vertical axis presents percentage of faults detected. Test case prioritization can have its impact on the fault detection. This is the important hypothesis that has been tested with the proposed tool and underlying methodologies. As the results revealed, the prioritized test cases were able to detect more faults. Thus the hypothesis has been tested and proved to be positive.

5. DISCUSSION

In this paper, our research work is focused on three aspects, 1. Automatic test suite generation, 2. Automatic discovery of dependency structures and 3. Test case prioritization. We have followed Automatic Test Suite approach to generate Chromosomes of the given input program. The whole test suite generation, with proposed methodology as illustrated in Figure 2, count produce comparable results with EvoSuite. Further the usage of GA operators identifies test cases which to be included in whole test suite. This is a representative and demonstrates high coverage of SUT. The mutation operators are used at Java byte code level to identify possible test cases. The results are obtained with several case study projects. Figure 4 shows different statistics of the case studies. The results presented in Figure 5 reveals the average number of missed infeasible targets between the proposed approach, EvoSuite and Single approaches. The proposed approach outperforms Single approach, as whole test suite is representative of multiple objectives and ensures high coverage. The proposed method is comparable with the EvoSuite with little difference in performance.

Apart from the above mentioned results, the average branch coverage for all case studies were observe with our method and compared with EvoSuite and Single. The rationale behind this is that the proposed approach and EvoSuite follows whole test suite generation concept that considers representation of multiple objectives so as to reduce the number of test cases besides ensuring high branch coverage. The results are presented, which in Figure 6 ,that reveals the aforementioned outcomes. The comparative results show that the proposed approach has slight performance than the EvoSuite.

Another significant contribution in this paper is automatic discovery of dependency structures. In the literature[7],[8] and [9],it is mentioned that dependency structures were used to prioritize test cases. As prioritization can improve the fault detection ratio, the dependencies of function calls in
terms of open and closed dependencies can influence the results. In this regard, we proposed a methodology and algorithm which extracts dependency structures automatically. In [8], the dependencies are manually extracted, our methodology automate the procedure of extracting dependencies. Since manual prioritization is very time taking, this is significant improvement in our research. Our algorithm ensures that the dependencies are automatically extracted and used to prioritize test cases. The prioritization is non-trivial and hard to achieve complexities in the case studies with thousands of branches. The process of extraction of dependencies is not simple with very complex projects. Therefore proposed methodology is extraction of dependencies automatically paves way for significant speed and performance in software industry with respect to testing. The proposed methodology for test case prioritization includes the process of extracting dependencies a blend of reflection API for method identity. By using execution traces, the call processing insights in runtime behaviour of SUTs. This knowhow paves way for automatic prioritization of test cases. Further this results in improving the fault detection ratio. The methodology is in Figure 3 and the results that shows the performance improvement are presented in Figure 9. In Figure 9(b), clearly shows test case prioritization performance improves significantly.

6. COMPARATIVE RESULTS
In this section, the two approaches results were compared with the TRO case study. The number of faults detected is compared with automatic discovery of dependency structures in the proposed methodology and the manual identification of dependency structures in [8]. Figure 10 shows the comparison results.

As shown in Figure 10, the performance of the two approaches for given SUT is recorded. The results reveals that, the proposed methodology, determines the dependency structures automatically which is comparable with that of [8], were dependency structures are manually identified and prioritization performed. The increased count value in the fault detection is due to the fact that the test case prioritization can help in running test cases in proper order that will reflect in the increase of fault detection ratio.

7. CONCLUSIONS AND FUTURE WORK
In this paper our focus is three aspects pertaining to software testing. First, we proposed and implemented a mechanism for generating automatic case suite generation with multiple
objectives besides achieving complete coverage. Second, we focused on the automatic extraction of dependency structures from SUT so as to improve prioritization of test cases. We proposed and implemented a methodology for this purpose. Third, we built a tool that demonstrates the automated test suite generation, automatic discovery of dependency structures and test case prioritization for improving the percentage of flaws detected. The integrated functionality has been tested. Our empirical results reveal significant improvement in the percentage of detection of flaws with the new approach. This research can be extended further to adapt our approaches to Software Product Lines (SPLs) in future. Software product line is the modern approach in software development that takes care of set of products with core and custom assets. The new products are derived based on the variabilities in the requirements. The SPL approach improves software development in different angles especially when software is delivered as product to clients. SPLs and their configuration management is complex in nature. Testing such applications need improved mechanisms that can leverage the characteristics of SPL for improving quality.

8. REFERENCES


C. Prakasa Rao & P. Govindaraju


C. Prakasa Rao & P. Govindaraju

