Automated Functional Test Case Prioritization
For Increased Rate of Fault Detection

K. Uma Maheshwar
Student
Department of Computer Science & Engineering
Jntua, Anantapuramu (515002), Andhra Pradesh, India

Dr. S. Vasundra
Professor
Department of Computer Science & Engineering
Jntua, Anantapuramu (515002), Andhra Pradesh, India

Abstract

Testing plays important role in Software Development Life Cycle. Test Case Prioritization explains how the Test Cases are executed in particular order to achieve certain goals specified by the clients. In previous Fine Grained technique is used to prioritize the test cases. This process is done in random order, due to this Fault Detection Rate is going to be decreased. Our main aim is to increase the fault detection rate, then only we can rectify errors as many as possible which leads to fast delivery of project. Here prioritization process is going to be done between Test cases. Due to this there is no proper order of execution among the test cases, that which test case should execute first. In this paper we are going to use Coarse Grained technique based on Functional coverage to perform prioritization among the test suites rather than test cases. Test suite is the collection of the test cases that are dependent on each other. We are focusing on Functional test suites which means how much extent the test suites are dependent on each other. In previous some authors performed prioritization technique on real time systems and they generated Datasets for those systems. We are going to use those Datasets as reference to our project and we are calculating DSP volume, DSP Height and APFD. With this output we can identify that which test suite should execute first and we can also reduce the repeated execution of test suites.

Keywords: Test Case Prioritization, Functional Test Suites, Automation, DSP, APFD, DSP Volume, DSP Height, Software Engineering, Testing, Debugging, Test Execution.

I. INTRODUCTION

Now-a-days bulky portion of software engineering projects normally involves testing. To increase the efficiency and accuracy of testing effort within partial possessions, test case prioritization can be performed. Test case prioritization is the process of organizing test cases in a order to accomplish a certain goal. The main target of test case prioritization is to increase the rate of fault detection; that is, to find the preponderance defects and as early as possible. Finding defects in advance can increase early defect fixing and finally lead to earlier delivery. Prioritization has usually been applied to test suites that take days or weeks to run; however, with agile development processes becoming more widespread in industry, the approaching for prioritization techniques to have an impact is increasing.

Earlier work on test case prioritization [1] demonstrates that prioritization techniques are effective for recuperating rate of fault detection. However, these approaches do not consider test suites that contain functional dependencies between tests. Functional dependencies are the connections and relationships among system functionality determining their run succession; for example, a function G can only be executed if some requirement holds and function F enables this precondition. As test cases emulate this functionality, they also succeed to these dependencies; therefore, executing some test cases requires executing other test cases first [9].

While many existing techniques could be correct to test suites containing functional dependencies, applying them requires new algorithms for computing the sequencing of tests to protect the dependencies [10].

II. BACKGROUND

A. Test Case Prioritization:

In several software projects, it is compulsory to execute test suites in order of priority to consume limited resources and time efficiently, and to increase the outlook of finding defects early [2] explanation of an industry team mate with a test suite that takes in the order of seven weeks to run. In such cases, prioritization is important.

Test case prioritization techniques plan test cases for execution in an order that attempts to increase their usefulness at meeting some recital goal. Various goals are possible; one involves rate of fault detection estimate how speedily faults are detected within
the testing process. An improved rate of fault detection during testing can provide quicker answer on the system under test and let software engineers begin correcting faults before than might otherwise be possible. One application of prioritization techniques involves regression testing—the retesting of software following modifications; in this background, prioritization techniques can take advantage of information gathered about the earlier execution of test cases to obtain test case orderings.

**Definition:**
Test case prioritization is the process of arranging test cases to be executed in a fastidious order so that test cases with a privileged priority are executed earlier in the test sequence. The priority is defined relative to some test criterion; for example, the number of code statements covered.

**Rate of Fault Detection:**
It is defined as the number of defects found in allocated time. If more no of defects are encountered then it is huge task to find them.

### III. EVOLUTION

In this section, we summarize an experimental assessment of our DSP techniques using six systems built toward industry use which are all at present in use as far as the authors are attentive. All six systems are used to calculate both open and closed dependency structures. The target of this appraisal is to ascertain whether dependency structure prioritization is able to increase the rate of fault detection relative to coarse-grained techniques and relative to existing random fine-grained techniques.

<table>
<thead>
<tr>
<th>Artifacts</th>
<th>LOC</th>
<th>Faults</th>
<th>Tests</th>
<th>Dependencies</th>
<th>Graph Density</th>
<th>Max Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELITE</td>
<td>2487</td>
<td>54</td>
<td>64</td>
<td>64</td>
<td>0.0317</td>
<td>17</td>
</tr>
<tr>
<td>GSM</td>
<td>385</td>
<td>10</td>
<td>51</td>
<td>65</td>
<td>0.0509</td>
<td>6</td>
</tr>
<tr>
<td>CRM</td>
<td>975</td>
<td>33</td>
<td>30</td>
<td>30</td>
<td>0.0689</td>
<td>6</td>
</tr>
<tr>
<td>MET</td>
<td>10674</td>
<td>37</td>
<td>81</td>
<td>80</td>
<td>0.0246</td>
<td>6</td>
</tr>
<tr>
<td>CZT</td>
<td>27246</td>
<td>27</td>
<td>548</td>
<td>341</td>
<td>0.0021</td>
<td>5</td>
</tr>
<tr>
<td>BASH</td>
<td>59800</td>
<td>3-4</td>
<td>1061</td>
<td>461</td>
<td>0.0009</td>
<td>12</td>
</tr>
</tbody>
</table>

### IV. PROPOSED SYSTEM

Here we are going to use Coarse Grained Technique for prioritizing the Test Suites. In this paper, we intend a set of new techniques for functional test case prioritization based on the inbuilt structure of dependencies between tests, which we call Dependency Structure Prioritization.

The techniques are divided into two categories: one based on Open dependency structures, in which a functional dependency between test cases \( t_1 \) and \( t_2 \) indicates that \( t_1 \) must be executed at some point earlier than \( t_2 \); and one based on closed dependency structures, in which a dependency between test cases \( t_1 \) and \( t_2 \) indicates that \( t_1 \) must be executed immediately before \( t_2 \). We classify two strengths to these techniques.

First, they are model-based they use only the information contained in the dependency structure which means that scheming the test order does not need information from earlier test runs, such as the amount of code each test covers. This allows prioritization on the first test run, as well as allowing up-to-date prioritization for an iteration before the tests from the previous iteration have finished executing.

Second, by maintaining fine-grained test suites, but executing complete scenarios, we realize a good balance between the problem of fine-grained versus coarse-grained tests. Fine-grained tests lead to more bendable test suites, and lead to less problems with cascading test failures, but coarse-grained tests have a higher level of fault detection, due to the interaction between the “inner” tests [3].

**A. Dependency Structure:**
A dependency structure is a directed acyclic graph (DAG), \( G=(V,E) \) in which \( V \) is a set of nodes, and \( E \) is a set of arcs between these nodes. In this work, the set \( V \) defines a set of test cases. The set \( E \) defines the functional dependencies between the tests.

**B. Open Dependency Structure**
An open dependency structure is one in which a dependency between two test cases \( t_1 \) and \( t_2 \) specifies that \( t_1 \) must be executed at some point before \( t_2 \), but not essentially without delay before \( t_2 \).
C. Closed Dependency Structure:
A closed dependency structure is one that is not open; that is, a dependency between two test cases t1 and t2 specifies that t1 must be executed immediately before t2 [4]. They split dependencies into three categories:
(1) Abstraction dependencies, which are dependencies based on hierarchical decomposition in a system model, such as aggregation and composition;
(2) Temporal dependencies, which are dependencies based on time that is, one scenario must be executed before the other temporally; and
(3) Causal dependencies, which are dependencies based on data or resources that is, some resource that is produced by scenario A is used by scenario B.

D. Proposed Algorithm:
Algorithm:
- Step1. Accept the input Test cases(Datasets)
- Step2. Automatically Calculate dependencies between test cases.
- Step3. Prioritization of test cases
  a) Prioritization for Open Dependency Structures.
     (1) Calculate DSP Volume By Using Algorithm 1.
     (2) Calculate DSP Height By Using Algorithm 2.
     (3) Calculate Test case ordering by using graph coverage value.
     (4) Graph coverage value is calculate by using Algorithm3 and Algorithm 4.
  b) Prioritization for Closed Dependency Structures.
     (1) Calculate DSP Sum by using formula given in paper.
     (2) Calculate DSP ratio by using formula given in paper.
     (3) Calculate DSP Sum/ratio by using formula given in paper.
- Step4. Analysis all the calculated values.
- Step5. Generate graphs with above calculated values.

V. PRIORITIZING TEST CASES BASED ON DEPENDENCY STRUCTURE
For open dependency structures, we define two ways to measure the graph coverage value of a test case based on a dependency structure:
(1) The total number of dependents of the test case, and
(2) The longest path of direct and indirect dependents of the test case.
Using these values, the priority of tests is calculated using a weighted depth-first search algorithm in which the next child selected in the search is defined by its graph coverage value.
For closed dependency structures, we define three ways to measure the graph coverage value. These measure the coverage of paths throughout the dependency structure. The three coverage measures for paths are:
(1) The number of non executed test cases in the path (recall that a test case may be executed more than once if they have more than one dependent);
(2) The ratio of non executed to executed test cases in the path, with a higher weighting for those test cases toward the end of the path, thereby giving higher priority to longer paths; and
(3) A combination of the above two: the number of non executed test cases divided by the height of the path.
VI. Prioritization for Open Dependency Structures

We define two graph coverage process based on open dependency structures: DSP volume and DSP height.

A. DSP Volume:
The DSP volume coverage measure gives a higher weight to those test cases that have more dependents. To calculate the DSP volume of a test case, one needs to calculate all direct and indirect dependents of that test case [5].

Algorithm 1

a) \textit{Input:}
G: an n x n Boolean adjacency matrix representing direct dependencies between test cases.

b) \textit{Output:}
D: an n x n Boolean adjacency matrix representing indirect dependencies between test cases.

\begin{verbatim}
for i = 1 to n do
    for j = 1 to n do
        if D[i,j] = 1 then
            for k = 1 to n do
                D[i,k] = max(D[i,k], D[i,j] V D[j,k])
            end for
        end if
    end for
end for
\end{verbatim}

B. DSP Height:
The DSP height coverage assess gives a higher weight to those test cases that have the innermost dependents. To calculate the DSP height of a test case, one needs to analyse the height of all paths from that test case, and take the length of the highest path as the weight. This can be done using a instantly forward depth-first search algorithm on the graph, which is based on the Floyd-Warshall shortest-path algorithm [6].

Algorithm 2

a) \textit{Input:}
G: an n x n Boolean adjacency matrix representing direct dependencies between test cases.

b) \textit{Output:}
D: an n x n integer adjacency matrix representing the length of indirect dependencies between test cases.

\begin{verbatim}
for i = 1 to n do
    for j = 1 to n do
        for k = 1 to n do
            D[i,j] = max(D[i,j], D[i,k] V D[k,j])
        end for
    end for
end for
\end{verbatim}

C. Calculating Test Case Orderings by Using Graph Coverage Value:

Algorithm 3

a) \textit{Input:}\nG: an n x n Boolean adjacency matrix representing direct dependencies between test cases.

b) \textit{Input:}\nw: a weight function mapping test cases to their graph coverage values.
c) Output:
T: a test suite prioritized by the graph coverage value.
(1) T = <-
(2) I = get independent tests(G)
(3) while I != 0 do
(4) i Belongs to t:I | (for All t Belongs to I w(t)>=w(t'))
(5) T = WEIGHTED_DFS_VISIT(I, G, w, T)
(6) I = I \{i
(7) end while
(8) return T

Algorithm 4
a) Input:
G: an n x n Boolean adjacency matrix representing direct dependencies between test cases.
b) Input:
v: the vertex from which to start the search
c) Input:
w: a weight function mapping test cases to their graph coverage values.
d) Input:
T: the tests that have been prioritized so far.
e) Output:
T: a test suite prioritized by the graph coverage value.
(1) T = T \_lv
(2) C = get children(G,v)
(3) while C != 0 do
(4) C Belongs to {t:C | for All t Belongs to C w(t) >= w(t')}
(5) if C Belongs to T then
(6) T = WEIGHTED_DFS_VISIT(c, G, w, T)
(7) end if
(8) C = C \{c
(9) end while
(10) return T

VII. PRIORITIZATION FOR CLOSED DEPENDENCY STRUCTURES

A set of paths is linearly independent if all paths contain at least one node that is not present in any other path in the set. To create a set of linearly independent paths, we use the adapted depth-first search algorithm proposed by Poole [7].

A. DSP Sum:
The DSP sum coverage evaluate gives a upper weight to paths that have more non executed test cases. To calculate the DSP sum of a path, one simply counts the number of non executed test cases in that path.

\[ \text{DSP}_{\text{sum}}(p) = \# \{ i \in 1..\#p | \neg \text{seen}(t_i) \} \]

B. DSP Ratio:
The DSP ratio coverage appraise gives a higher weight to paths that have a higher ratio of non executed tests to executed tests, while also giving weight to longer paths. To estimate the DSP ratio of a path, one first calculates the prejudiced sum of the path in which the weight of a test case is its index in the path if it has not been executed.

\[ \text{DSP}_{\text{ratio}}(p) = \sum_{1}^{\#p} w(t_i)/\#p \]

C. DSP Sum/Ratio:
The DSP sum/ratio treatment is simply the number of non executed test cases alienated by the elevation of the path. Therefore, this is similar to the DSP ratio, except it does not weight the strength of test cases.

\[ \text{DSP}_{\text{sum/ratio}}(p) = \# \{ i \in 1.....\#p | \neg \text{seen}(t_i) \} /\#p \]
VIII. MEASURES:

A. Average Rate of Fault Detection:
The first dependent variable of the Experiments is the average rate of fault detection for each of the test suite orderings. This measures how quickly a test suite detects faults. APFD is measured as a percentage, with higher values implying a quicker rate of fault detection [8].

\[
APFD = 1 - \frac{\sum TF_1 + TF_2 + \cdots + TF_m}{n m} - \frac{1}{2n}
\]

where, \( m \) is the number of exposed faults, \( n \) is the total number of test cases and \( TF_i \) is the position of first test case in \( T \) that exposes fault \( i \).

B. Number of Test Cases Required to Find All Faults:
The second measure attempts to mitigate the problem of assuming a uniform cost for executing each test suite in the APFD metric. The cost of executing a coarse-grained test suite is more than that of a fine-grained test suite due to the redundant tests, and the APFD measure does not take this into account.

IX. RELATED WORK:
Numerous researchers have provided solutions for test case prioritization for increasing the fault detection rate, and others have looked at the use of dependencies in testing. However, none have looked at the combination of the two. In this section, we delineate the work most applicable to ours.

X. CONCLUSION
With help of Six systems we performed prioritization and Dependency among Functional test suites Automatically and achieved certain goals specified by the client. We reduced the time taken for executing the repeated test suites, due to this the fault detection rate has been increased when compare to previous technique.

XI. FUTURE WORK
Presently we are using normal algorithms for prioritizing and maintaining dependency structures between the test suites. In future we are planning to use Genetic Algorithms for implementing Code coverage technique for maintaining prioritization and dependency. This technique includes the following aspects:

1. Requirement coverage,
2. Total requirement coverage,
3. Additional requirement coverage and
4. Statement coverage

XII. RESULTS

A. Existing System Graph:
B. Proposed System Graph:

![Graph showing comparison of Real Time Systems with Email and Database]  

REFERENCES