Application Testability for Fault Detection Using Dependency Structure Algorithm

[1]Shah Ubaid Nisar, [2] T. S. Shiny. Angel
[1]M.Tech Student, [2]Assistant Professor (O.G)
Department of Software Engineering, SRM University, Chennai, India
[1]ubaidshah20swe@gmail.com, [2]angel_d_leno@yahoo.com

***Abstract*—The purpose of this paper is to provide prioritization techniques which are effective for improving rate of fault detection. In this paper dependency structure algorithms are used to make application testability more efficient and reliable. The priority is based on a graph coverage value. Test case prioritization techniques organize the test cases in a test suite by ordering such that the most beneficial are executed first thus allowing for an increase in the effectiveness of testing. One of the performance goals i.e. the fault detection rate, is a measure of how quickly faults are detected during the testing process. Previous work on test cases demonstrates system based evaluation but however these approaches do not consider application based testability. The nature of the techniques preserves the dependencies in the test ordering. Experimental evaluation on two applications indicates that our techniques offer a solution to the prioritization problem in the presence of test cases with dependencies. Two applications are tested over different code and shows good results than the existing one. The results demonstrate a cost-effective technique. The results of both experiments provide clear evidence that the dependency structure algorithms have potential to be used to prioritize test suites. Based on our experimental results and analysis, we can also comment upon our expectations for how well prioritization may perform on programs that are much larger. Also, large programs in which the output values are very sensitive to each individual computation are likely to show greater improvement with our approach as well.**

***Index Terms*— test suite, testing and debugging, fault analysis**

1. INTRODUCTION

Software testing can be stated as the process of validating and verifying that a computer program, application, product. Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. A test case, in software engineering, is a set of conditions or variables under which a tester will determine whether an application, software system or one of its features is working as it was originally established for it to do. A test case prioritization schedule test case in order to increase their ability to meet some performance goals, one goal is to increase the rate of fault detection. Increasing the rate of fault detection provides defect fixing, fault fixing, and meets requirements early. The purpose of this paper is to provide prioritization techniques which are effective for improving rate of fault detection. In this paper dependency structure algorithms are used to make application testability more efficient and reliable. Previous work on test cases demonstrates system based evaluation but however these approaches do not consider application based testability.
The author defines two ways to measure dependencies (Weighted Volume and Deepest Height). Several related techniques for functional test case prioritization based on dependency structures. The goal of this evaluation is to establish whether Dependency Algorithms are able to increase the rate of fault detection or not. Experimental evaluation on two applications indicates that our techniques offer a solution to the prioritization problem in the presence of test cases with dependencies. Two applications are tested over different code and shows good results than the existing one. The results from the two applications indicate that our techniques offer a solution to the prioritization problem in the presence of test cases with dependencies

Prioritization of Test Case Ordering

Prioritization is the process of scheduling test cases to be executed in a particular order so that test cases with a higher priority are executed earlier in the test sequence. Test runner will run prioritized test cases got from the Height and Volume. Test cases are run then based on the priority and completes there process and give output to the further test cases [5].

Functional Dependency

Functional dependencies are the interactions among functionality of a system. The sequencing in scenarios represents the order in which the interactions take place. Put simply, some interactions cannot occur until and unless some other interactions occur first [5].

$$\begin{tikzcd}
 & & A1 \arrow[dl] \arrow[dr] & & & A2 \arrow[d] & \\
 & B3 \arrow[dl] \arrow[dr] & & B4 \arrow[dr] & & B5 \arrow[dl] \arrow[d] \arrow[dr] & \\
B6 & & B7 & & B8 & B9 & B10
\end{tikzcd}$$

Figure.1 Dependency structure

Fig. 1 depicts an example dependency structure. In this structure, A1 and A2 have no dependencies, while nodes B3 to B10 are dependent on other nodes; for example, B3 is dependent on A1. Note the difference between direct and indirect dependencies: B6 is directly dependent upon B3 and indirectly dependent upon A1.

Open and Closed Dependency Structures

An open dependency structure is one in which a dependency between two test cases t1 and t2 specifies that t1 must be executed at some point before t2, but not necessarily immediately before t2. In other words, once t1 has been executed, the dependent node remains open for execution, irrelevant of any other nodes being executed. For example, in Fig. 1, if node A1 is executed, then nodes B3 and B4 are available. If A2 is then executed, nodes B3, B4, and B5 are all available to be executed.
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. For example, if all dependencies in Fig. 1 are closed dependencies and nodes A1 and A2 are executed in order, then to execute B3 or B4, node A1 would need to be executed again. Some dependency structures may contain a mix of both open and closed dependencies [5].

2. RELATED WORK

Experiments with test case prioritizations using relevant slices
Several techniques have been developed to prioritize the execution of existing test cases to expose faults early during the regression testing process. In this paper, author present a new approach to prioritize test cases that takes into account the coverage requirements present in the relevant slices of the outputs of test cases. The author have implemented three different heuristics based on our relevant slicing based approach to prioritize test cases and conducted experiments to compare the effectiveness of our techniques with those of the traditional techniques that only account for the total requirement coverage.
A random test case prioritization
In this paper, the author not only proposes a new family of coverage-based ART techniques, but also shows empirically that they are statistically superior to the RT-based technique in detecting faults. Regression testing assures changed programs against unintended amendments. Rearranging the execution order of test cases is a key idea to improve their effectiveness. The author proposes a set of ART prioritization techniques guided by white-box coverage information. The author also conducts an empirical study to evaluate their effectiveness.
Clustering test cases to achieve effectiveness and scalable prioritization incorporating expert knowledge
In this paper, the author introduces 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. By clustering test cases, based on their dynamic runtime behavior, we can reduce the required number of pair-wise comparisons significantly. The author presents an empirical study that shows that the resulting prioritization is more effective than existing coverage-based prioritization techniques in terms of rate of fault detection.
A case study in model based testing of specification and implementation
The method is based on a test graph, which provides a partial model of the application under test. The test graph is used in combination with an animator to generate test sequences for testing the formal specification.
In specification-based testing (of implementations), the specification defines the behavior to which the implementation should conform. The process of testing is then verifying whether or not the specification and the implementation define the same behavior.
Using dependency structure for prioritization of functional test suite
Test case prioritization is the process of ordering the execution of test cases to achieve a certain goal, such as increasing the rate of fault detection. Increasing the rate of fault detection can provide earlier feedback to system developers, improving fault fixing activity and, ultimately, software delivery. The author present several related techniques for functional test case prioritization based on dependency structures. The author calls this family of prioritization techniques Dependency Structure Prioritization (DSP).The nature of the techniques preserves the dependencies in the test ordering. The hypothesis of this work is that dependencies between tests are representative of interactions in the system under test, and executing complex interactions earlier is likely to increase the fault detection rate, compared to arbitrary test orderings. Empirical evaluations on three systems built toward industry use demonstrate that these techniques increase the rate of fault detection compared to the rates achieved by the untreated order, random orders, and test suites ordered using existing "coarse-grained" techniques based on function coverage.

3. PROPOSED SYSTEM

The author defines two ways to measure dependencies (Weighted Volume and Deepest Height). Several related techniques for functional test case prioritization based on dependency structures.

Input
The input can be a method. The input can be a class .The input can be package or library. The input can be test case or test suite (Collection of Test case).

Generate test cases with the help of test tool
The Junit test tool is used to generate test case. The test cases generated based on the source code and flow control.

Using Deepest Height and Weighted Volume Algorithms
Deepest Height and Weighted Volume used to find priority; Priority based on open and closed dependency structure.

Prioritization of Test Case Ordering
Test runner will run prioritized test cases got from the deepest height and weighted volume. Test cases are run then based on the priority and completes there process and give output to the further test cases.

PROBLEM ADDRESSED
The purpose of this paper is to provide prioritization techniques which are effective for improving rate of fault detection. In this paper dependency structure algorithms are used to make application testability more efficient and reliable. Previous work on test cases demonstrates system based evaluation but however these approaches do not consider application based testability.
The results from the two applications indicate that our techniques offer a solution to the prioritization problem in the presence of test cases with dependencies. Two applications are tested over different code and shows good results than the existing one.

ALGORITHM
The author defines two algorithms Weighted volume dependents and Deepest height dependents. Height and Volume algorithm is used to find the priority. The priority is based on open dependency and closed dependency.

Weighted Volume Dependents
Weighted Volume Dependent gives higher weight to those test cases that have more dependents.
To calculate volume, calculate all direct and indirect dependents of that test case. In this algorithm, if a direct edge exists between two nodes, m and n, then replacing row m with row n will result in the direct dependents of n becoming dependents on m.

Algorithm 1. VOLUME
Input U: a × a direct dependencies.
Output V: a× a indirect dependencies

```
1: for n є 1...p
2: for m є 1...p
3: if A [n, m] = 1
4: for r є 1…p
5: A [n, r] = A [n, r] ˅ A [m, r]
6: end
7: return V
```

Deepest Height Dependents
Deepest Height Dependent gives a higher weight to those test cases that have the deepest dependents.
To calculate Height, calculate the height of all paths from that test case, and take the length of the longest path as the weight.

Algorithm 2. HEIGHT
Input U: a × a dependencies between test cases.
Output V: a ×a length of indirect dependencies between test cases.

```
1: for n є 1…p
2: for m є 1…p
3: for r є 1…p
4: A [n, m] = max A [n, m], A [n, r] + A[r, m]
5: end
6: return V
```

The author defines two algorithms based on closed dependency structure: sum, and ratio. These two prioritization techniques provide weights to paths in the dependency structure.
Sum
The Sum gives a higher weight to paths that have non-executed test cases. To find a balance between longer paths.

To calculate sum of a path, count the number of non-executed test cases in that path.

Ratio

The Ratio gives a higher weight to paths that have a higher ratio of non- executed test to executed tests, while also giving weight to longer paths.
To calculate ratio first calculates the weighted sum of the paths in which the weight of a test case is its index in the path if it has not been executed.

4. EXPERIMENTAL EVALUATION

The goal of this evaluation is to establish whether Dependency Algorithms are able to increase the rate of fault detection or not.
Experimental evaluation on two applications indicates that our techniques offer a solution to the prioritization problem in the presence of test cases with dependencies.
Two applications are tested over different code and shows good results than the existing one.

ANALYSIS

The author presents experimental analysis on two applications.

MAIL APPLICATION

The mail application for sending mail consists of some data sets such as: The mail id of the sender and receiver, Subject of mail, Files to be attached and the body or content of the mail. The mail application dependency consists of some of the dependency data sets have to be checked by test analysis.
The dependencies of the mail application system such as: Total no of faults, No of test analysis, Test type.
The analysis of the mail application system can be done with fault analysis and analysis based on dependencies and maximum.

DATABASE APPLICATION

The database application scenario consists of the data sets such as: User name, Age, City, Gender, Mail id are those data sets to be registered.
The DB application system can also be tested with some of the dependency constraints to be checked and prioritized.
The dependency of the data sets prints the set of test results such as: Total no of faults, No of test analysis, Test type, Total no of dependencies.

5. RESULTS

The author presents the experimental results on two different applications.
The results from the two applications indicate that our techniques offer a solution to the prioritization problem in the presence of test cases with dependencies.
Two applications are tested over different code and shows good results than the existing one.
The two applications demonstrate that prioritization of test cases increase the rate of fault detection, fault fixing, and finds defects early.

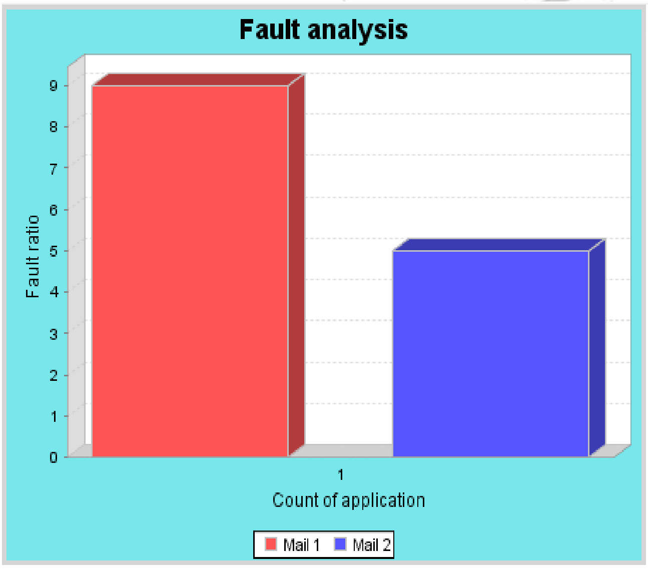

Figure.2 Fault analysis

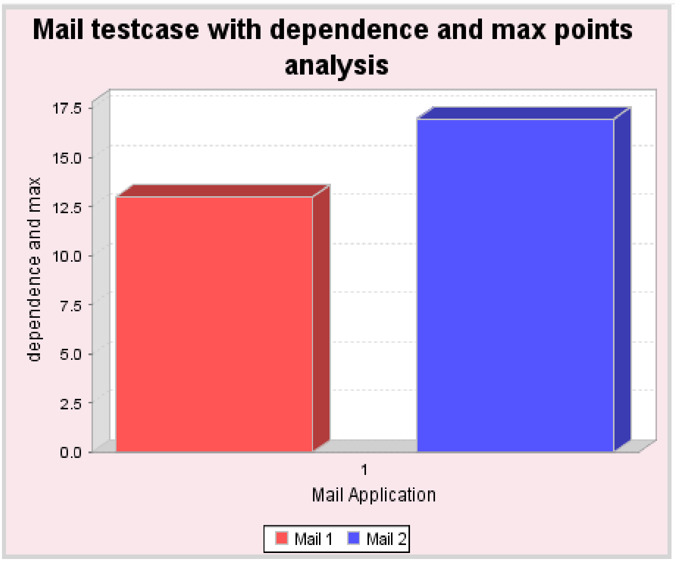

Figure.3 Mail test case

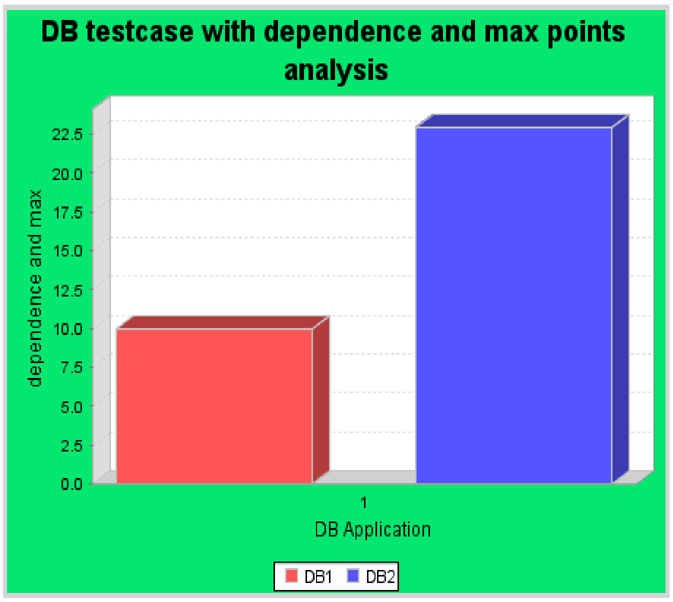

Figure.4 DB test case

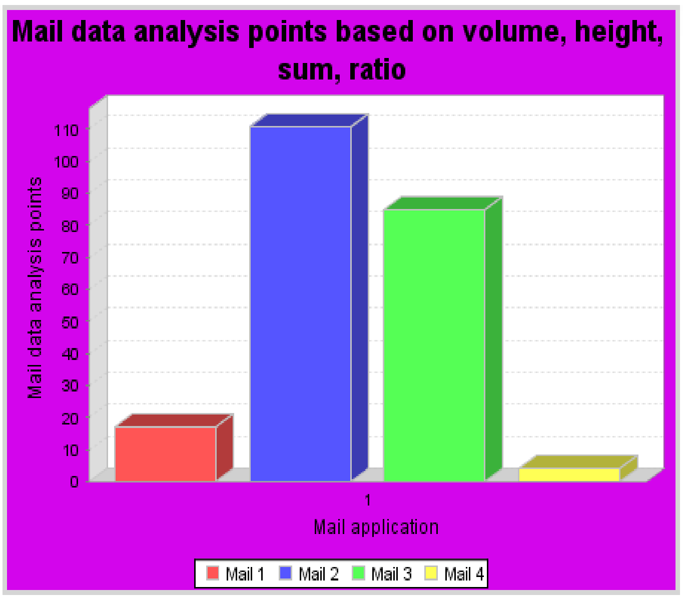

Figure.5 Mail data analysis based on Volume, Height, Sum, Ratio

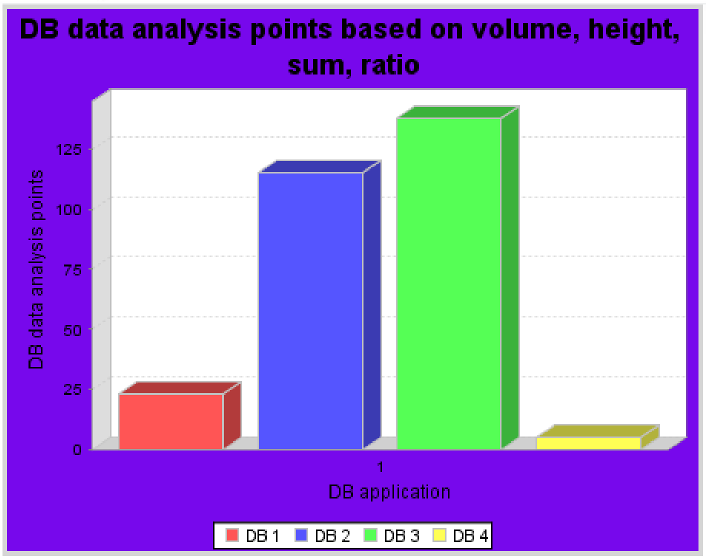

Figure.5 DB data analysis based on Volume, Height, Sum, Ratio

6. CONCLUSION AND FUTURE WORK

The results from the two applications indicate that our techniques are able to increase the rate of fault detection. The results demonstrate a cost-effective technique. The results of both experiments provide clear evidence that the dependency structure prioritization methods have potential to be used to prioritize test suites

In future, addition to assessing the techniques on more systems, we can also plan to compare our techniques with other types of prioritization techniques.
Based on our experimental results and analysis, we can also comment upon our expectations for how well prioritization may perform on programs that are much larger. Also, large programs in which the output values are very sensitive to each individual computation are likely to show greater improvement with our approach as well. On the other hand, large programs whose output values are not very sensitive to small changes in computation.

7. REFERENCES

[1] S. Yoo, M. Harman, P. Tonella, and A. Sus.”Clustering test cases to achieve effectiveness and scalable prioritization incorporating expert knowledge.”proc. 18th int’l symp. Software testing and analysis, pp. 201-212, 2009.
[2] D. Jeffrey and N. Gupta, “Experiments with Test Case Prioritization Using Relevant Slices,” J. Systems and Software, vol. 81, no. 2,pp. 196-221, 2008.
[3] B. Jiang, Z. Zhang, W. Chan, and T. Tse, “Adaptive random test case prioritization,” IEEE/ACM Int’l Conf. Automated Software Eng., pp. 233-244, 2009.

[4] T. Miller and P. Strooper. “A case study in model based testing of specification and implementation.”Software Testing, Verification, and Reliability, vol. 22, no. 1, pp. 33-63 2012.

[5] Shifa-e-Zehra Haidry and Tim Miller “Using Dependency Structures for Prioritization of Functional Test Suites.” IEEE transactions on software engineering, vol. 39, no. 2,2013.