# Interactive Multiplots For Comparing Coverage Effectiveness

Adam M. Smith* Joshua J. Geiger†

University of Pittsburgh

## Abstract

Software testing increases confidence in the correctness of application source code. Using different orderings of a test suite can enable the earlier detection of defects which allows developers to start fixing them sooner. Finding the best ordering is difficult and there are many approaches that prioritize the ordering, resulting in many possible test suite orderings to choose from. This paper presents a tool that allows the user to visually examine the results of several test suite prioritization techniques and to use interaction to gain insights about the properties of the test suite and its different orders.

## 1 Introduction

Inevitably, errors are made while designing and implementing software systems. Software developers execute tests $\{t_1, t_2, t_3, \ldots, t_n\}$ in a test suite $T$ to isolate defects and gain confidence in the correctness of the code. Each test case in the test suite exercises specific points in the system, comparing the actual output of the code to the hand computed expected one. If a test fails, then a defect is present in the area that the test executes. As the source code grows in size and number of features, new tests are written for the new functionality. These new tests, however, do not obviate the old ones. To ensure that the new features do not cause the system to regress, developers include every previously written test in the collection of tests. This process of executing and re-executing the entire test suite is known as regression testing.

Gradually, adding new tests and retaining old ones increases the size of the test suite until its execution time can become prohibitively expensive. In some cases the regression test suite runs for several weeks [2]. One method of altering the test suite to resolve this issue is test suite prioritization [5]. Test suite prioritization attempts to find a reordering of the test cases that is more likely to locate defaults earlier in the execution of the test suite without risking the loss of coverage by removing a test. The tests are ordered based on certain criteria that are obtained during a process called coverage monitoring.

Coverage monitoring measures the code coverage of a test case $t_i$. Code coverage describes any metric that enumerates specific points in the source code that are executed when a test is run, whether they are a line, a block, a method, a branch [6], or some other type. Each specific program point is called a requirement. Given a test suite $T$, coverage monitoring gives a set of requirements $R(T) = \langle r_1, r_2, \ldots, r_m \rangle$. Each individual test $t_i$ is associated with a subset of requirements $R(t_i) \subseteq R(T)$. Each test is said to cover its associated set of requirements.

Figure 1 shows an example of a test suite with 4 tests and 5 requirements. An X in a cell represents that the test for that row covers the requirement in that column. Imagine that the shown test suite is run in its original order. In this case all of the requirements are not covered until 8 time units have passed. Conversely, if the test suite is executed in reverse order all of the requirements are

| | $r_1$ | $r_2$ | $r_3$ | $r_4$ | $r_5$ | Execution Time |
|---|---|---|---|---|---|---|
| $t_1$ | X | X | X | X | | 4 |
| $t_2$ | | | X | X | | 1 |
| $t_3$ | | X | | | | 1 |
| $t_4$ | X | | | | X | 2 |

Figure 1: Example Test Suite

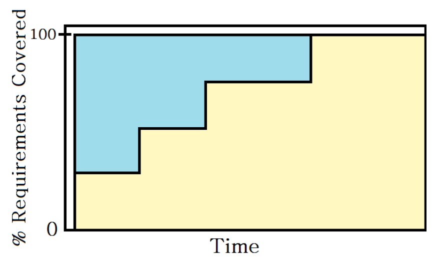


Figure 2: Coverage Effectiveness

covered in 4 time units. Covering all of the requirements sooner gives a better chance to find faults sooner so that the developers can begin to make changes at an earlier time.

The metric coverage effectiveness [4] was developed in order to rate a test suite prioritization. CE is derived from the step function of the cumulative coverage of a test suite as shown in Figure 1. Each test suite offers the possibility of covering more total requirements. So when the cumulative coverage is plotted against time a step function is formed. CE itself is calculated by dividing the area under the step function for the actual order by the area under the curve of a test suite that covers all of its requirements instantly. This value is between 0 and 1 where 0 means that no requirements were covered, and 1 means that all of the requirements were covered instantly.

## 2 Motivation

There are a total of $n!$ possible orderings of a test suite where $n$ is the number of tests. Clearly for most test suites these cannot all be enumerated to find ordering with the highest CE. For this reason there are several algorithmic ways to find good orderings, including random sampling. However, it is difficult to interpret the results of several algorithms or random samples by reading text alone. Therefore, this project aimed to create a visualization that can allow software testers to instantly see how algorithmic prioritizers perform and also to examine a set of random random prioritizations simultaneously. There is related work in the field of visuzaling test suites [3] [1], however, they focus on fault localization or other features of a test suite.

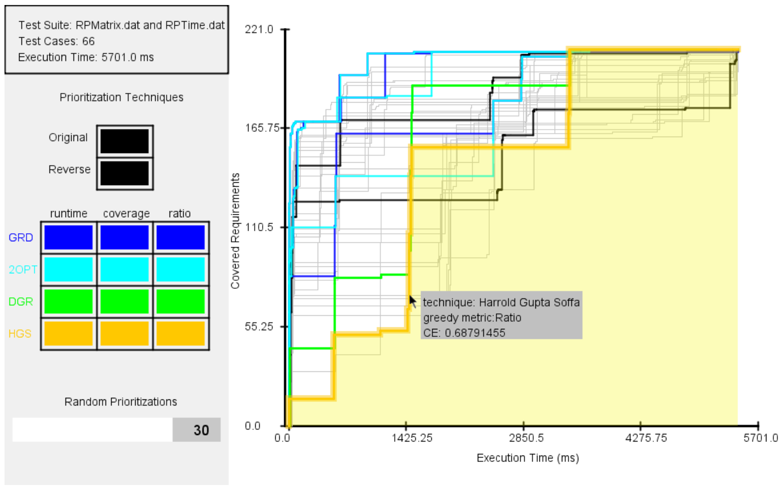
## Acknowledgements

The authors wish to thank Gregory Kapfhammer, Manos Renieris, and Liz Marai for assistance with this project.

## References

[1] M. Breugelmans and B. Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites.

*e-mail: ams292@cs.pitt.edu
†e-mail:jjg???@cs.pitt.edu


Test Suite: RPMatrix.dat and RPTime.dat
Test Cases: 66
Execution Time: 5701.0 ms

technique: Harrold Gupta Soffa
greedy metric:Ratio
CE: 0.68791455

[2] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Test Case Prioritization: A Family of Empirical Studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[3] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, New York, NY, USA, 2002. ACM.

[4] G. M. Kapfhammer and M. L. Soffa. Using coverage effectiveness to evaluate test suite prioritizations. In *Proceedings of the ACM Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, November 2007.

[5] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[6] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.