Test Suite Reduction and Prioritization with Call Trees

Adam Smith, Joshua Geiger, and Gregory M. Kapfhammer
Department of Computer Science
Allegheny College
gkapfham@allegheny.edu

Mary Lou Soffa
Department of Computer Science
University of Virginia
soffa@cs.virginia.edu

ABSTRACT

This paper presents a tool that (i) constructs tree-based models of a program's behavior during testing and (ii) employs these trees while reordering and reducing a test suite. Using either a dynamic call tree or a calling context tree, the test reduction component identifies a subset of the original tests that covers the same call tree paths. The prioritization technique reorders a test suite so that it covers the call tree paths more rapidly than the initial test ordering. In support of program and test suite understanding, the tool also visualizes the call trees and the coverage relationships. For a chosen case study application, the experimental results show that call tree construction only increases testing time by 13%. In comparison to the original test suite, the experiments show that (i) a prioritized suite achieves coverage much faster and (ii) a reduced test suite contains 45% fewer tests and consumes 82% less time.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Languages, Verification

Keywords

regression testing, call trees

1. INTRODUCTION

Modern object-oriented programs exhibit complex patterns of behavior during testing and execution. A *call tree* contains nodes and edges that represent a program's method invocations. A *dynamic call tree* (DCT) includes a node for each method call, preserving full execution context at the expense of having unbounded depth and breadth. Alternatively, a *calling context tree* (CCT) has bounded depth and breadth because it coalesces nodes and uses back edges when methods are recursively or iteratively invoked [1]. Even though the DCT and CCT are normally used for program profiling, recent approaches to regression testing use call trees to reduce a test suite as well [4, 5].

This paper describes a comprehensive framework that builds and analyzes call trees in order to perform both test suite reduction and prioritization. The collection of testing components includes a call tree constructor that instruments the program under test with probes to create a DCT or a CCT. In an effort to control both the size and execution time of a test suite, the reduction technique identifies a subset of the original tests that covers the same call tree paths. The prioritization tool reorders a test suite so that it covers the tree paths more effectively than the initial test ordering. The tool also visualizes the call trees and the coverage relationships so that it is easier to understand the run-time behavior of the program and the tests.

We distinguish our tool from prior testing techniques that use call trees (e.g., [4, 5]) because our regression tester performs reduction and prioritization for object-oriented programs. In contrast, McMaster and Memon focus on reducing the test suites for procedural and graphical user interface (GUI) applications. The current implementation contains five algorithms that perform both reduction and prioritization: Harrold, Gupta, Soffa (HGS) [2], overlap-aware greedy [9], non-overlap-aware greedy [7], delayed greedy [8], and k-way greedy [3]. Our testing framework enhances these previously developed schemes by considering each test's execution time. A reduced test suite is characterized by how well it decreases both testing time and the total number of tests. The framework evaluates a prioritization according to a metric called *coverage effectiveness*.

2. REGRESSION TESTING TOOL

Figure 1 illustrates the use of call trees to reduce and prioritize a test suite (a grey background highlights the modules that are important contributions). Currently, the tool analyzes JUnit 3.8.1 test suites and programs written in the Java 1.5 programming language. The call tree constructor uses either static or dynamic instrumentation techniques to insert probes into the program under test. These probes execute before and after all of the methods and tests in order to build the call tree. We implemented the call tree constructor with the Java 1.5 and AspectJ 1.5 programming languages. The call tree construction procedure uses aspect-oriented *pointcuts* and before and after *advice* in order to construct either a DCT or a CCT. The tree constructor also employs aspects to (i) initialize the call tree before the first test case runs, (ii) store the tree prior to the conclusion of testing, and (iii) measure the execution time of each test.

The tool builds a call tree that contains a node for every test case invocation that occurs during testing. Each path under a test case node is a unique test requirement because it represents a series of method calls that took place during testing. After the creation of the call tree, a reduction algo-

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.
ACM 978-1-59593-882-4/07/0011.

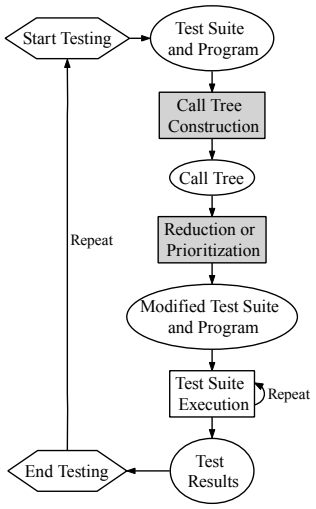

Figure 1: Reduction and Prioritization Tool.

rithm analyzes this tree in order to produced a modified test suite that is guaranteed to cover all tree paths with (hopefully) fewer test cases. The HGS reducer analyzes the test coverage information and initially selects all of the tests that cover a single requirement [2]. In the next iteration, HGS examines all of the requirements that are covered by two tests and it selects the test case with the greatest coverage. HGS continues to select tests until it obtains a minimized suite that covers all of the tree paths.

The overlap-aware greedy reducer uses the approximation algorithm for the minimal set cover problem [9]. Greedy reduction with overlap awareness iteratively selects the most cost effective test case for inclusion in the reduced test suite (i.e., evaluating each test according to the ratio of time to coverage means that low values indicate good cost effectiveness). During every successive iteration, the overlap-aware greedy algorithm re-calculates the cost effectiveness for each leftover test according to how well it covers the remaining test requirements. This reduction technique terminates when the reduced test suite covers all of the call tree paths that the initial tests cover. The k-way greedy algorithm operates in an analogous manner except that it considers every possible group of k tests during each iteration [3].

The delayed greedy approach proceeds in a similar fashion while also exploiting information concerning both the requirements that a test case covers and the tests that cover a specific call tree path [8]. Since each of these reduction methods leaves the excess tests in the initial test suite, the prioritization scheme identifies a test reordering by repeatedly reducing the residual tests. The prioritizer's invocation of the overlap-aware reducer continues until the original suite of tests is empty. The non-overlap-aware prioritizer sorts the tests by cost, coverage, or cost effectiveness [7]. When provided with a target size for the reduced test suite, the non-overlap-aware reducer selects from the sorted tests until the modified test suite reaches the size limit (unlike the other approaches to reduction, this method does not guarantee the coverage of every call tree path).

The testing tool supports repetition at two distinct locations, as evidenced in Figure 1. The same modified test suite can be leveraged whenever either the execution environment is different or the changes to the program under test are minimal. If the program modifications are likely

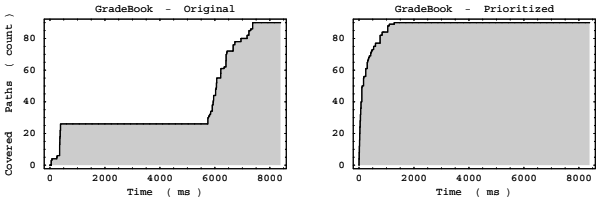

Figure 2: Coverage Functions.

to result in execution behavior that is significantly dissimilar from past behavior, then the entire testing process can be repeated. When evaluating the quality of the original and prioritized test suites, the framework uses a coverage function that shows how the tests cover the tree paths over time. As shown in Figure 2, a point on a coverage function curve corresponds to the number of tree paths covered at that time. A test suite's coverage effectiveness (CE) is the ratio between the area under its coverage function and the coverage area of an ideal test suite that immediately covers all of the paths. The value of CE falls inclusively between 0 and 1, with a high value indicating a high quality test suite.

Due to space constraints, we focus on the reduction and prioritization of the test suite for a `GradeBook` application containing 1455 non-commented source statements (NCSS), 147 methods, and 10 classes. The experiments reveal that the call tree construction probes increase test suite execution time by 12.3%. When using the overlap-aware greedy algorithm, reduction decreases test suite size by 45% and testing time by 82%. The results also demonstrate that the coverage effectiveness of the original test suite was .38 while the prioritized test suite achieves a CE value of .96. The coverage function plots in Figure 2 reveal that the prioritized tests cover the ninety unique call tree paths much more rapidly than the original suite. In summary, the experiments suggest that this tool supports efficient and effective regression testing. In future work, we intend to incorporate additional reduction and prioritization algorithms and include new evaluation metrics such as the average percentage of faults detected (APFD) [6]. Evaluation of the tool will continue as we test additional case study applications.

3. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc of PLDI*, pages 85–96, 1997.

[2] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[3] Z. Li, M. Harman, and R. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[4] S. McMaster and A. Memon. Call stack coverage for test suite reduction. In *Proc of 21st ICSM*, pages 539–548, 2005.

[5] S. McMaster and A. Memon. Call stack coverage for GUI test-suite reduction. In *Proc of 17th ISSRE*, pages 33–44, 2006.

[6] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[7] M. Rummel, G. M. Kapfhammer, and A. Thall. Towards the prioritization of regression test suites with data flow information. In *Proc of 20th SAC*, pages 1499–1504, 2005.

[8] S. Tallam and N. Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *Proc of 6th PASTE*, pages 35–42, 2005.

[9] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.