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

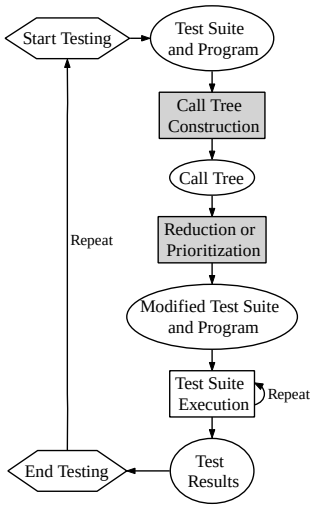


Figure 1: Reduction and Prioritization Tool.

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

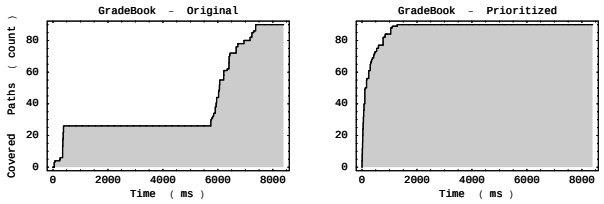


Figure 2: Coverage Functions.

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.

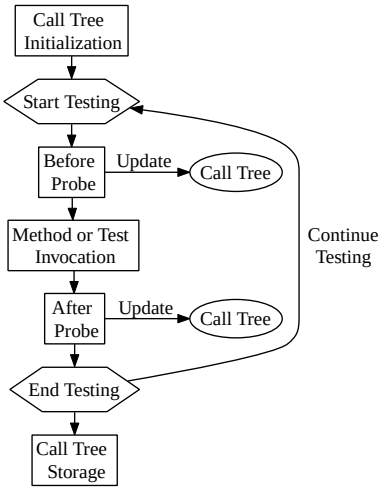


Figure 3: Call Tree Construction Probes.

APPENDIX

A. DEMONSTRATION OVERVIEW

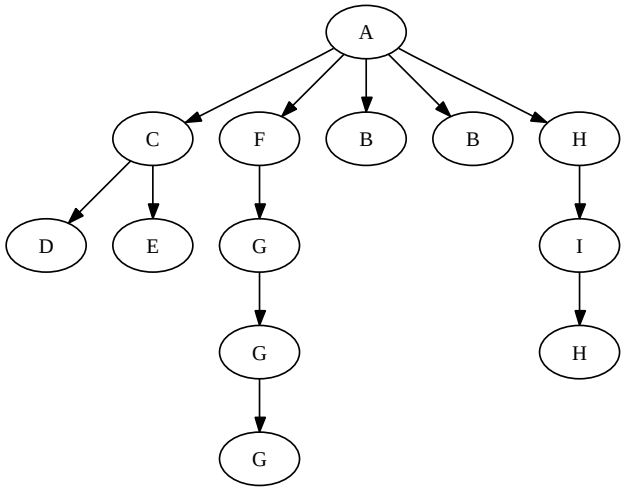
The demonstration will use slides, animations, and full color graphics in order to explain call trees and regression testing. The slides will motivate the need for call tree-based reduction and prioritization. During the demonstration, we will show how the instrumentation probes construct a dynamic call tree and a calling context tree. We will furnish animations that explain how the reduction and prioritization algorithms create the modified test suite. The presentation will also include insights into the goals that guided the design and implementation of the testing framework.

During the testing of several case study applications, we will explain the relevant configurations and commands. Finally, the presentation will conclude by reviewing the experimental results that highlight the trade-offs in the efficiency and effectiveness of the regression testing techniques. We will point out important empirical trends with appropriate data visualizations such as bar charts and scatter plots. Throughout the demonstration, we will regularly use visualizations (e.g., call trees, coverage reports, and graphs) that were automatically generated by the testing framework.

B. REGRESSION TESTING DETAILS

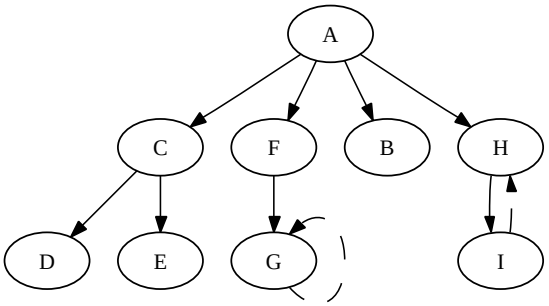
B.1 Instrumentation Techniques

The presentation will furnish insights into our approach to assembling call trees with aspect-oriented programming (AOP) techniques and AspectJ. The call tree constructor uses instrumentation probes that are inserted in either a static or a dynamic fashion. The static instrumentor places the probes into the program under test before test suite execution whereas dynamic instrumentation inserts the probes during testing. The static instrumentor can operate in a batch mode that inserts the probes into multiple applications during a single run. Static instrumentation must occur each time the program under test changes. The load-time dynamic instrumentor introduces the probes on a per-class basis with either the JVM tools interface (JVMTI) or a custom class loader. The use of dynamic instrumentation improves the flexibility of testing at the cost of a potential increase in the time overhead of test suite execution.



Number of Nodes = 13, Number of Edges = 12

(a)



Number of Nodes = 9, Number of Edges = 10

(b)

Figure 4: Examples of the (a) DCT and (b) CCT.

B.2 Call Trees

Using Figure 3, the demonstration will illustrate how the probes build a call tree. Upon completing the call tree's initialization, the tree constructor executes a probe before and after the execution of each method and test case. The testing tool can construct either a DCT or a CCT. The DCT records the complete execution context while incurring low probe overhead and moderate to high tree storage costs. Alternatively, the CCT reduces tree space overhead at the expense of slightly increasing the execution time of the probes. Even though the CCT coalesces certain nodes in order to minimize space overhead, it still preserves all unique method calling contexts.

Figure 4(a) provides an example of a DCT with thirteen nodes and twelve edges. In this tree a node with the label "A" corresponds to the invocation of the method A and the edge A $\rightarrow$ B indicates that method A invokes method B. The existence of the two DCT edges A $\rightarrow$ B reveals that method A repeatedly invokes method B. In the example from Figure 4(a), the DCT represents the recursive invo-

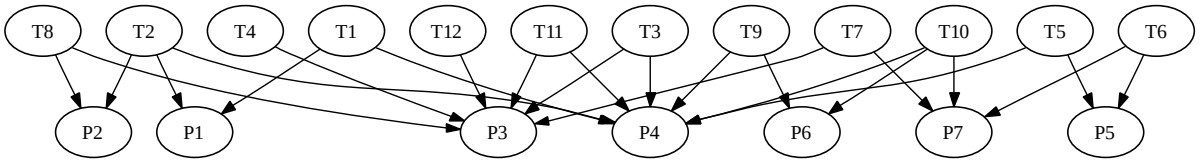


Figure 5: Overlap in the Coverage of Call Tree Paths.

cation of method G by chaining together edges of the form G → G. The CCT in Figure 4(b) coalesces the DCT nodes and yields a 30.8% reduction in the number of nodes and a 16.7% decrease in the number of edges. For example, the CCT combines the two B nodes in the DCT into a single node. The CCT also coalesces nodes and introduces back edges when a method calls itself recursively (e.g., the DCT path G → G → G) or a method is repeatedly executed (e.g., the DCT path H → I → H). Figure 4(b) shows that the tool depicts a CCT back edge with a dashed line.[1]

B.3 Reduction and Prioritization Examples

A test suite often consists of test cases that overlap in their coverage of the unique call tree paths. The presentation will use Figure 5 to show how seven separate test requirements are covered by twelve unique test cases. The testing tool automatically analyzes the call tree in order to construct this type of tree-based summary of the coverage relationships. In this tree, a directed edge from a test case T_i to a tree path P_j indicates that P_j is covered by T_i (or, that T_i *covers* P_j). For example, an edge between test case T_8 and path P_2 indicates that P_2 is covered by T_8 (alternatively, T_8 covers P_2). Since the test suite in Figure 5 contains a significant amount of overlap in test requirement coverage, it is a candidate for reduction. Inspection of Figure 5 reveals that executing a reduced test suite containing T_2, T_3, T_6, and T_9 instead of the original twelve tests will still cover all of the seven paths (other reductions are also possible for this test suite).

Using graphical and tabular examples, the demonstration will also explain each of the reduction and prioritization techniques. For example, suppose that the non-overlap-aware greedy prioritizer reorders a test suite $T = \langle T_1, T_2, T_3 \rangle$, as described in Figure 6(a). Reordering T according to cost gives the initial ordering $\langle T_1, T_2, T_3 \rangle$ since T_1 consumes one time unit and T_2 and T_3 both consume two time units (for this example, we resolve ties by creating the order $\langle T_i, T_k \rangle$ when $i \leq k$ or $\langle T_k, T_i \rangle$ if $i > k$). Prioritization according to tree path coverage yields the ordering $\langle T_3, T_1, T_2 \rangle$ because T_3 covers six requirements and T_1 and T_2 both cover five. Figure 6(b) also shows that prioritization by the cost to coverage ratio creates the ordering $\langle T_1, T_3, T_2 \rangle$.

B.4 Coverage Effectiveness

Using a different test suite $T = \langle T_1, T_2, T_3 \rangle$ that covers a total of five call tree paths, the presentation will illustrate how the tool calculates coverage effectiveness. Figure 7 shows that test T_2 takes ten seconds to execute while

[1]The introduction of one or more back edges into a CCT forms cycle(s). Even though a CCT is not strictly a tree, the tree edges are distinguishable from the back edges [1].

Test Case	Cost	Coverage	Ratio
T_1	1	5	1/5
T_2	2	5	2/5
T_3	2	6	2/6

(a)

Effectiveness Metric	Test Case Order
Cost	T_1, T_2, T_3
Coverage	T_3, T_1, T_2
Ratio	T_1, T_3, T_2

(b)

Figure 6: The Cost and Coverage of a Test Suite.

Test Case	Execution Time (sec)
T_1	5
T_2	10
T_3	4

Total Testing Time = 19 seconds

Figure 7: Test Suite Execution Time.

T_1 and T_3 respectively consume five and four seconds during test suite execution. In this example, we assume that test T_2 covers four requirements and T_3 and T_1 cover three and two requirements, respectively. There are $3! = 3 \times 2 \times 1 = 6$ different orderings in which we could execute this simple test suite. In order to characterize the coverage effectiveness of a test suite prioritization, Figure 8 plots the cumulative number of covered test requirements during the execution of T. The shaded area under these coverage curves highlights the effectiveness of a test ordering (i.e., a large shaded area suggests that an ordering is highly effective).

We construct a coverage function with the assumption that the tool marks a tree path as covered when one of the path's covering test cases terminates. For example, Figure 8(a) shows that the execution of T_1 leads to the coverage of two test requirements (i.e., P_1 and P_2) after five seconds of test suite execution. Under the assumption that an ideal test suite immediately covers all of the test requirements, we define the coverage effectiveness of a test suite as the ratio between its coverage area and the ideal coverage area.

The demonstration will employ the example in Figures 7 through 10 to demonstrate that prioritization can improve the effectiveness of testing. For example, we see that the test suite ordering $T = \langle T_1, T_2, T_3 \rangle$ yields a coverage area of 36 and an effectiveness value of only .3789. Yet, visual

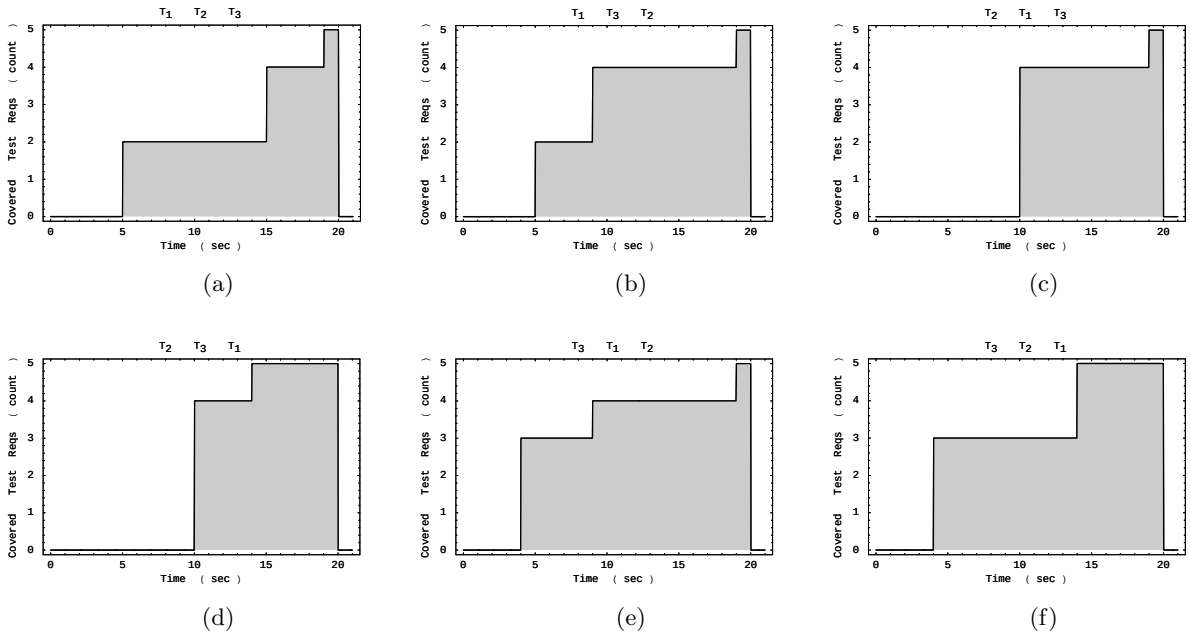


Figure 8: Coverage Effectiveness of Regression Test Suite Prioritizations.

inspection of the plots in Figures 8(e) and (f) suggest that the test orderings $T = \langle T_3, T_1, T_2 \rangle$ and $T = \langle T_3, T_2, T_1 \rangle$ both cover the requirements faster than any of the other orderings. Indeed, Figure 10 shows that these orderings have a coverage effectiveness of .5789. The results also show that different test orderings can lead to the same effectiveness value. For example, Figure 10 indicates that the orderings $T = \langle T_1, T_2, T_3 \rangle$ and $T = \langle T_2, T_1, T_3 \rangle$ have the same low coverage effectiveness.

B.5 Design and Implementation Goals

Following the lead of successful software engineering tools (e.g., Daikon [2]), we designed and implemented the testing framework as a series of command-line tools. The presentation will discuss how we incorporated existing free and open source software so that we could focus on the implementation and evaluation of the core testing techniques. For example, the tool uses Graphviz [3] to show the call trees and the coverage relationships. The framework leverages the R package [4] to visualize the efficiency and effectiveness results with bar charts and scatter plots. R is also used to (i) calculate the CE values for a test ordering and (ii) perform a statistical analysis of the results with hypothesis testing and linear regression. The tool executes a test suite with the JUnit framework. We implemented the testing techniques with Java and used the eXtensible Markup Language (XML) for all of the configuration files.

C. USING THE TESTING TOOL

The demonstration will also include a walk through of the relevant configuration files and command-lines for running the testing components. First, we will run the original test suite and note its execution time. Next, we will execute the test suite while performing call tree construction. After storing the call tree, we will use a module that analyzes the tree and invokes a reduction or prioritization algorithm. Figure 11 provides an excerpt from the configuration file for the regression testing component. In this example, the tool stores the call tree in an XML file called `tree.TestGradeBook.xml.cct`. This configuration will reduce `GradeBook`'s test suite with the overlap-aware greedy algorithm. The `useActualTestCosts` tag indicates that the time overhead for running each test case should be used during reduction. Upon completion of the reduction procedure, the demonstration will run the reduced test suite and show the decrease in test suite execution time. Similar steps will be taken to demonstrate the use of prioritization.

Test Case	Call Tree Paths				
	P_1	P_2	P_3	P_4	P_5
T_1	✓	✓			
T_2	✓	✓	✓		✓
T_3	✓			✓	✓

Figure 9: Path Coverage for a Test Suite.

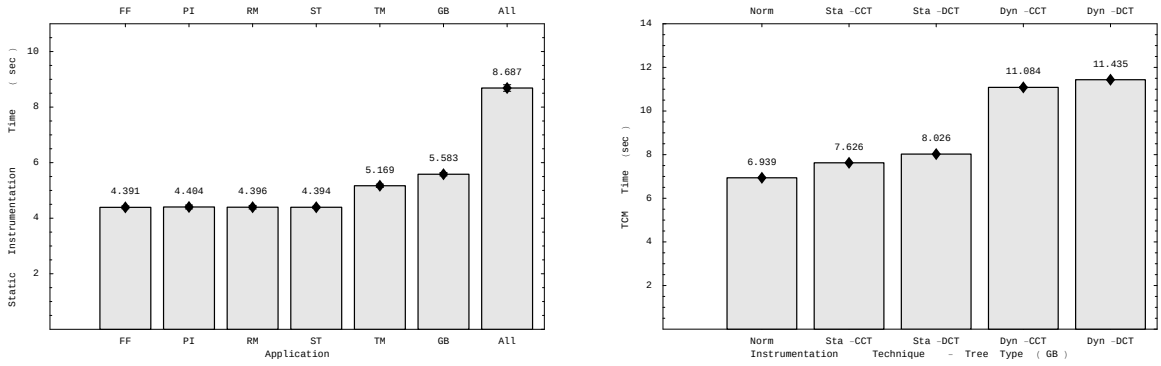
Ordering	Cov Area	Cov Effectiveness
$T_1\ T_2\ T_3$	36	.3789
$T_1\ T_3\ T_2$	48	.5053
$T_2\ T_1\ T_3$	36	.3789
$T_2\ T_3\ T_1$	41	.4316
$T_3\ T_1\ T_2$	55	.5789
$T_3\ T_2\ T_1$	55	.5789

Ideal Coverage Area = 95

Figure 10: Test Coverage Effectiveness.

D. EMPIRICAL EVALUATION

We conducted experiments with the intent of measuring the (i) time overhead of statically inserting the probes, (ii) impact that static instrumentation has on the space overhead of an application, (iii) time overhead associated with assembling the call trees, and (iv) benefits associated with



(a) (b)

Figure 12: The Costs of Static and Dynamic Instrumentation.

```
<PerformReductionConfiguration>
  <callTreeName>tree.TestGradeBook.xml.cct
  </callTreeName>
  <performType>ReduceGreedy</performType>
  <useOverlap>true</useOverlap>
  <useActualTestCosts>true</useActualTestCosts>
</PerformReductionConfiguration>
```

Figure 11: Excerpt from a Configuration File.

reduction and prioritization. The empirical study used six Java applications that range in size from 548 to 1455 non-commented source statements (NCSS). We always executed the static instrumentor and the call tree constructor in ten separate trials for each case study application and we report an arithmetic mean and a standard deviation. We performed all of the experiments on a GNU/Linux workstation with kernel 2.6.11-1.1369, a dual core 3.0 GHz Pentium IV processor with 1 MB of L1 cache, and 2 GB of memory.

In this appendix, we focus on analyzing the costs associated with constructing the call trees. During the demonstration, we will also furnish additional graphs about the performance of the reduction and prioritization components. Figure 12(a) presents the time overhead associated with statically instrumenting the case study applications. Ten executions of the instrumentor on the small `StudentTracker` (`ST` - 622 NCSS) application yields a mean instrumentation time of 4.4 seconds with a standard deviation of .04 seconds. Figure 12 reveals that the static instrumentation of a large case study application such as `GradeBook` takes 5.6 seconds. Since we can instrument all six of the applications in less than 9 seconds, the results in Figure 12(a) also suggest that the batch mode is efficient.

Using Figure 12(b), the demonstration will compare the costs of constructing the call tree with different types of instrumentation techniques. In this graph, "Norm" labels the time required to execute `GradeBook` without instrumentation. The other bars in this graph show how the variation of the instrumentation technique and the type of the call tree impacts test suite execution time. The results indicate that testing time only increases by 12.3% when the tool uses statically inserted probes to create a CCT. It is also efficient to assemble a DCT with static instrumentation since this approach only raises the time overhead to about 8 seconds. Finally, Figure 12(b) reveals that the flexibility of dynamically inserting the probes comes at a cost (i.e., the Dyn-CCT and Dyn-DCT configurations cause testing time to increase by 59.7% and 64.7%, respectively).

E. CONCLUSIONS

During the conclusion of the demonstration, we will review the important contributions of our regression testing tool. In particular, our comprehensive framework is the first to integrate a call tree constructor with five important algorithms for performing test suite reduction and prioritization. Since the tool handles JUnit test suites, we judge that it will be very valuable to software engineering practitioners. The testing tool will also be useful to researchers because it contains modules that analyze and visualize the empirical results. We intend to release the entire testing framework under a free and open source license. We will make the source code, binary distribution, and documentation of our regression testing tool available for download after the Automated Software Engineering (ASE) 2007 conference. Please consult the following Web site for more information about the regression testing framework:

`http://cs.allegheny.edu/~gkapfham/research/kanonizo/`

Acknowledgments Robert S. Roos and Andrew Thall provided valuable suggestions that improved the presentation and content of the paper.

F. APPENDIX 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. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2006.

[3] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[4] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.