Call Stack Coverage for GUI Test-Suite Reduction

Scott McMaster and Atif Memon
Department of Computer Science, University of Maryland, College Park, Maryland
{scottmcm,atif}@cs.umd.edu

Abstract

Graphical user interfaces (GUIs) are used as front-ends to most of today's software applications; testing GUIs for functional correctness is needed to ensure the overall correctness of these applications. The event-driven nature of GUIs presents new challenges for testing. One important challenge is test suite reduction. Conventional reduction techniques/tools based on static analysis are not easily applicable due to the increased use of multi-language GUI implementations, callbacks for event handlers, virtual function calls, reflection, and multi-threading. Moreover, many existing techniques ignore event handlers from libraries, and fail to consider the context in which a handler executes. Consequently, they yield GUI test suites with seriously impaired fault-detection ability. This paper presents a new reduction technique based on the ***call stack coverage criterion****. Call stacks may be collected for any executing program with very little overhead. An empirical study involving three large GUI-based applications shows that call stack based reduction provides an excellent tradeoff between reduction in test suite size and loss of fault-detection effectiveness.*

1. Introduction

Users increasingly interact with modern software through graphical user interfaces (GUIs). Testing GUIs for functional correctness is extremely important because (1) GUI code makes up an increasingly large percentage of overall application code and (2) due to the GUI's proximity to the end user, GUI defects can drastically affect the user's impression of the overall quality of a system.

Because of these factors, automated test case generation techniques for GUIs have been developed [12]. *Event-flow coverage* based test-case generation has been shown to be effective for defect detection in GUI applications [13]. However, the number of tests generated by using event flow coverage can be quite large. An event-flow-adequate test suite may be too large to fully execute regularly in a rapid development and integration environment that mandates, for example, nightly builds and smoke tests.

Test suite reduction [[5][19][16]] (also referred to as *test suite minimization* in the literature) seeks to reduce the number of test cases in a test suite while retaining a high percentage of the original suite's fault detection effectiveness. Several techniques are available which employ algorithms to reduce the size of a test suite while maintaining coverage adequacy relative to some criterion. For example, our earlier work [9] presented *call stack coverage* as a practical and effective basis for performing test suite reduction, advancing the state-of-the-art for coverage-based test suite reduction. Traditionally, these techniques, including our own, have been evaluated against conventional, non-GUI software.

The event-driven execution model for a GUI differs from that of other types of software. A given piece of code in a GUI application may be executed in many different *contexts* due to the increased degrees of freedom that modern GUIs provide to users. The context may be essential to uncovering defects; yet most existing coverage criteria are not capable of capturing context. Furthermore, today's sophisticated GUI applications increasingly integrate multiple source-code languages and object code formats, along with virtual function calls, reflection, multi-threading, and event handler callbacks. These features severely impair the applicability of techniques that rely on static analysis or the availability of language- and/or format-specific instrumentation tools. Thus, we believe that GUI-intensive software poses new challenges for coverage-based testing that require the development of new solutions.

In previous work [9], we presented the call-stack coverage criterion for single-threaded, non-event-driven applications. In this work, we extend our definition of call stack coverage to address new challenges introduced by modern GUI applications. Unlike criteria such as line (statement) or branch coverage, call stack coverage has the benefit of encapsulating valuable context information. And unlike other traditional coverage criteria such as dataflow/def-use pairs [15], call stack coverage is easily captured in a multi-language application, and with or without the availability of source code. We apply our modified technique to three multi-threaded GUI applications written in Java. We present an empirical evaluation showing that call stack coverage-based test suite reduction produces better results for GUI applications compared to traditional techniques.

Contributions: This paper makes the following contributions to the field of test-suite reduction:

1. Extends our previous definition of call stack coverage to account for multi-threaded runtime environments.
2. Discusses practical considerations for the implementation of call stack collection in the domain of modern GUI applications.
3. Empirically evaluates call stacks versus several traditional coverage criteria for use in test suite reduction in modern GUI applications.

The remainder of the paper is structured as follows: In the next section, we provide definitions and a brief summary of prior work using call stack coverage for test suite reduction. Section 3 discusses our solutions to specific problems that arise in modern GUI applications. In Section 4, we provide details about the implementation of our tool suite. Section 5 describes our empirical evaluation and presents results. In Section 6, we survey related work. We discuss our conclusions and future directions in Section 7.

2. Background

In this section, we present some background and definitions relating to test suite reduction and call stacks. Additional details may be found in [9].

2.1. Test Suite Reduction

Informally, the goal of *test suite reduction* is to reduce the size of a given test suite while preserving as much of its fault-detecting ability as possible. Most approaches to this problem are based on eliminating test cases that are redundant relative to some coverage criterion, such as program-flow graph edges [16], dataflow [19], or dynamic program invariants [4]. The test suite reduction problem is closely related to *test case prioritization* [2], because any reduction technique can be turned into a prioritization technique by repeated application of the reduction algorithm to the remainder of the suite. In our previous work, we proposed the coverage of *call stacks* as an effective new criterion to apply to test suite reduction.

2.2. Call Stacks

A *call stack* is the sequence of active calls associated with each thread in a stack-based architecture. Methods are pushed onto the stack when they are called, and popped when they return or when an exception is thrown (where supported, as in Java or C++). An example of a call stack from a simple Java program appears in Figure 1.

```
(Ljava/lang/Object;ILjava/lang/Object;II)V
Ljava/lang/System;arraycopy
([BII)V Ljava/io/BufferedOutputStream;write
([BII)V Ljava/io/PrintStream;write
()V Lsun/nio/cs/StreamEncoder$CharsetSE;writeBytes
()V Lsun/nio/cs/StreamEncoder$CharsetSE;implFlushBuffer
()V Lsun/nio/cs/StreamEncoder;flushBuffer
()V Ljava/io/OutputStreamWriter;flushBuffer
()V Ljava/io/PrintStream;newLine
(Ljava/lang/String;)V Ljava/io/PrintStream;println
([Ljava/lang/String;)V LHelloWorldApp;main
```

Figure 1: Example Call Stack in Java

Each method may be represented using its name, the types of its parameters, and return value. Thus a call stack is an ordered sequence of named methods.

An efficient data structure for recording call stacks on a given thread is the *calling context tree*, or CCT [1]. In a CCT, each node represents a method, the root is the entry point of the thread, and each edge represents a method call.

3. Call Stacks in Modern GUI Applications

GUI applications present a number of interesting challenges for call-stack-coverage-based analysis, including multithreading, heavy use of third-party libraries and frameworks, object-oriented language features, and management of coverage data.

3.1. Multithreading

Our earlier work focused on a conventional single-threaded application written in C, which required us to monitor and record the state of a single stack. However, modern GUI applications are all multithreaded. Indeed, all *Java* applications are multithreaded, if for no other reason than the presence of the garbage collector. Here we extend our call stack coverage technique from [9] to make it practical and useful in GUI applications.

Definitions: Each running thread in a multi-threaded application has a *current stack* of active method calls, where the most recently called method is at the *top* of the stack. Each thread generates a set of current stacks over its lifetime. Let the set of all unique stacks generated by a thread t be denoted as $C(t)$. If $c = <m_1, m_2, \ldots m_n>$ is a call stack of depth n, we define a *substack* c_s (denoted by a subscript s) and a *superstack* c^s (denoted by a superscript s) as the following ordered sequences, which are themselves call stacks:

$$(1)\ c_s = <m_1, m_2, \ldots m_i>, i < n$$

$$(2)\ c^s = <m_1, m_2, \ldots m_n, \ldots m_i>, i > n$$

For a given call stack c in any thread t, there is associated with c a set of substacks $C(t)_s$ and a set of superstacks $C(t)^s$. We define the set of deepest, or *maximum depth*, stacks $C(t)_{max}$ in a thread t as follows:

$$(3)\ C(t)_{max} = \{c \in C(t) \mid C^s = \varnothing\}$$

where $\varnothing$ is the empty set. Since each maximum depth stack implies the existence of all of its substacks in C, C_{max} is a more compact representation of the set of all unique call stacks generated by a thread.

To characterize the behavior of an entire multi-threaded program, we combine call stack observations made on each thread that took part in a given program execution. We define the set of threads that existed during execution:

$$(4)\ T = <t_1, t_2, \ldots t_n>$$

The set of unique call stacks for a given program execution χ is:

$$(5)\ \chi = \cup \{ C(t)_{max} \mid t \in T \}$$

χ is the set of maximum-depth stacks observed on each thread, and each element of χ is a coverage requirement in our reduction technique. Note that the definition of χ allows for the possibility that a maximum-depth stack on one thread is a substack of a maximum-depth stack on another, and both stacks would appear in χ. Therefore, χ is *not* necessarily a set of *unique maximum-depth* stacks. Although this may cause our technique to produce less size reduction than it might otherwise, we allow this for practical reasons, as checking for substack relationships across all stacks in every $C(t)_{max}$ is very computationally expensive and of marginal benefit.

Implementation: Our implementation approach to collecting unique call stacks is to create a separate CCT for each thread as it is created, and then maintain that CCT over the thread's lifetime as methods are entered and exited. Ignoring recursion for the time being, traversing each path to a leaf in the CCT gives us precisely the set of unique maximum-depth call stacks needed for our analysis. When a thread exits, its CCT is traversed to calculate the set of unique call stacks seen on that stack, and the unique stacks are synchronously merged into a master list of unique stacks seen on all threads. This approach allows for greater application concurrency than the alternative, which is a single CCT shared and maintained by all threads. A potential drawback is that an application with many short-lived threads may stall frequently for processing of the CCTs, but this was not an issue in our studies.

3.2. Libraries and Frameworks

Another important factor not fully addressed in our original work is the use of third-party libraries and frameworks. Libraries and frameworks are essential to modern software development in general and GUI applications in particular. Many test coverage techniques only collect coverage element data based on instrumentation of first-party application source or object code. The reasons for this include the unavailability of necessary third-party source code and the impracticality under most techniques of instrumenting an entire large framework such as the Java 2 SDK. By making this tradeoff, coverage techniques potentially overlook vast amounts of interesting behavior induced in library code by the application.

For example, consider the program in Figure 2a. If no library code is instrumented, every execution of this program against integral input will satisfy statement, branch, and dataflow coverage. Thus, when used in test suite reduction, each of those coverage approaches would potentially drop all tests that exercise the code with integral input greater than or less than zero, thereby missing the array-index-out-of-bounds exception that occurs with such input.

In contrast, our call stack coverage technique includes the library calls that appear on application-generated call stacks. Therefore, it preserves at least one test that displays the abnormal control flow triggered by the exception. In this work, we aim to further show empirically that this information is both valuable and practically obtained. In general, writing a tool to collect call stack coverage information only requires method entry and exit hooks, which already exist on most compilers or runtime platforms to enable the construction of call profilers.

a)

```
public class ArrayTest {
        public static void main(String args[]) {
                String[] strings = {"first"};
                int index = Integer.parseInt( args[0] );
                System.out.println( strings[ index ] );
        }
}
```

b)

```
import java.lang.reflect.*;
public class ReflectionTest {
        public static void main(String args[])
                throws ClassNotFoundException,
                NoSuchMethodException,
                SecurityException,
                IllegalAccessException,
                InvocationTargetException
        {
                if( args.length != 2 ||
                  !(args[0].equals("toUpperCase") ||
                    args[0].equals("toLowerCase")) ) {
                      throw new IllegalArgumentException();
                }
                String command = args[0];
                Class str = Class.forName( "java.lang.String" );
                Method m = str.getMethod( command, null );
                Object result = m.invoke( args[1], null );
                System.out.println( result.toString() );
        }
}
```

c)

```
public class HelloWorldApp {
   public static void main(String[] args) {
      System.out.println("Hello World!");
   }
}
```

Figure 2: Sample Programs for Call Stack Analysis

3.3. Object-Oriented Language Features

Our prior work focused on a procedural program written in C. Modern GUI application frameworks – usually implemented in languages like C++, Java, and C# -- make extensive use of object-oriented language features such as virtual function calls, reflection, and callbacks for event handlers. It is not possible in general to statically determine which methods will be invoked by a program execution. Dynamic analysis based on call stacks is ideal in such an environment because the stack contains the actual methods invoked in all cases.

Consider the program in Figure 2b, which takes two parameters: A method name presumed to be *toUpperCase* or *toLowerCase,* and a string argument to pass to the method via reflection. The call stacks generated by various executions of this program will differ based on the method name parameter, which is clearly behavior that should be captured for the purposes of test suite reduction. But the use of reflection makes determining this statically impossible. Modern GUI and server applications are often built using frameworks that employ reflection-based component models where the types and methods to be used are not known until runtime. Call stacks are ideal for recording test coverage in reflection scenarios.

3.4. Coverage Data Size

In our earlier work with the *space* application [17], we observed 453 unique maximum-depth call stacks which became our test coverage requirements. Due to heavy use of libraries and the runtime environment itself, even an extremely simple Java application may generate thousands of call stacks. Indeed, in the version of Java used in this work, the simple program in Figure 2c generated 803 call stacks; a GUI application built with Java Swing such as one of our subject applications can easily generate hundreds of thousands. We will show in this paper that call stack data collection and test suite reduction remains feasible in that realistic environment.

4. Tools for Call Stack Coverage and Test Suite Reduction

4.1. Collecting Call Stacks

To illustrate our technique's ability to work without source-level instrumentation, we built a Java Virtual Machine Tool Interface (JVMTI) agent to collect the CCT data necessary for a call stack coverage analysis.

We made use of the JVMTI hooks for method entry and method exit to maintain a CCT for each thread. Recursive invocations are permitted in our tool but are only captured to a depth of one. As threads die and at the end of an execution, the coverage information from each CCT is merged and processed into a set of unique call stacks which are finally written to the file system. Qualitatively, the applications we use as our experimental subjects remain quite responsive in the presence of this instrumentation. This suggests that call stack coverage may be practical to capture in certain fielded GUI applications, which may be a subject of future work.

Since we collect coverage for each thread, we are by definition collecting data on system threads where the subject program is not even on the stack. Since activity on system threads (such as the one on which the garbage collector runs, or the one that pumps GUI events in the Java Swing libraries) is somewhat environmentally dependent and may vary from run to run, this introduces a potential element of non-determinism into our data collection and, by consequence, which tests we select in the reduction process. However, this could be considered a positive result, as certain test cases may be more likely than others to induce fault-indicating activity on the aforementioned system threads.

The output of our JVMTI agent consists of two files: The first file represents the observed call stacks as a list of tab-delimited method identifiers. We store Java Native Interface (JNI) method identifiers instead of full method signatures in order to save space. However, method identifiers are assigned by the JVM and are not necessarily consistent across different executions of the same program. So our second output file contains a map of JNI method identifiers to the full method signatures. When calculating the set of unique call stacks across two or more test cases, we use the maps to create a canonical form based on the method signatures.

4.2. Reducing Test Suites

As in our previous work, we use a C# implementation of the *ReduceTestSuite* heuristic presented in [5]. In our implementation, *ReduceTestSuite* begins by including all test cases that cover a single call stack. Then it picks a test case that covers the most call stacks from the subsets of cases with the next lowest cardinality, marking all of the subsets that contain this case. This process occurs repeatedly for higher cardinality subsets until all subsets are marked and, therefore, all call stacks are covered. For an analysis of the runtime of this algorithm, see [5].

5. Experiments

We implemented the call stack collection and reduction algorithms and ran two experiments to evaluate our test suite reduction technique.

5.1. Research Questions

We sought to evaluate the call stack reduction technique in terms of the size and fault detection effectiveness of the resulting test suites. Specifically, we wanted to directly compare the call-stack based technique to reduction based on four different types of coverage: event (E1), event-pairs (E2), line (statement) (L), and method (M). Line coverage has probably the widest support of any coverage technique among commercial and open source tools due to its balance between precision and practicality. Event coverage [13] is specially tailored for use in GUI applications, which can be modeled as sequences of events. In E1, each event in isolation is a coverage requirement, while in E2, unique pairs of events are included as coverage requirements. We also wanted to investigate whether test suites created by call stack reduction preserved more fault-detecting ability than randomly reduced suites of the same size. To that end, we designed two experiments that we present next: (1) Experiment 1, in which we compared call stack based reduction with event, event-pair, line, and method-based reduction, and (2) Experiment 2, in which we compared call stack reduction to randomly selected suites of the same size.

5.2 Subject Applications

We used three applications from the TerpOffice Suite [14] as our experimental subjects. TerpOffice is a business productivity suite written in Java by senior software engineering students over a period of years. The three applications we study are TerpPaint (TP), TerpWord (TW), and TerpSpreadsheet (TS). Table 1 shows key metrics for these applications' test suites. Each application is associated with a large universe of test cases generated from the event flow criterion [10], a set of single-fault versions, and a set of faults known to be detected by each test case.

Application	**TerpPaint (TP)**	**TerpWord (TW)**	**TerpSpreadsheet (TS)**
Test Universe Size	1500	1000	1000
# Detectable Faults (Versions)	43	18	101

Table 1: TerpOffice Applications

5.3. Measured Variables

As in [9], we measured fault detection effectiveness on a per-test-suite basis, i.e., two test suites were considered to be equally effective at detecting a specific fault if they each contain at least one case that exposes the fault. This is the approach adopted in [19]. For each reduction experiment, we captured the percentage size reduction:

(1) $100 * (1 - \text{Size}_{\text{Reduced}} / \text{Size}_{\text{Full}})$

And percentage fault detection reduction:

(2) $100 * (1 - \text{FaultsDetected}_{\text{Reduced}} / \text{FaultsDetected}_{\text{Full}})$

Since we dealt with a fairly small number of discrete faults in our experiments, we took averages of these quantities over large numbers of suites.

5.4. Threats to Validity

Threats to external validity are factors that may impact our ability to generalize our results to other situations. Our main threat to external validity in this study is the small sample size. In this study, we only run our data collection and test suite reduction process on three programs, which we chose for their availability. These programs were constructed in a similar manner and may not be representative of the broader population of programs. An experiment that would be more readily generalized would include multiple programs of different sizes and from different domains. Additionally, we would expect the effectiveness of the call stack minimization process to vary depending on aspects of the programming style used in the target application. In particular, when the application is composed of many small functions, call stacks provide finer-grained dynamic state information. Our subject applications are GUI-event-driven and thus contain many small event-handling methods. This should increase the effectiveness of our minimization technique relative to what it could do against an application that implemented the same behavior using relatively fewer or more monolithic functions. (Consider the pathological case where a program is composed of a single large function, which would have but a single call stack for all executions.) Finally, characteristics of original test suites (such as their fault detecting ability and how they were constructed) play a role in the size and fault detection reduction results. This threat can be addressed in future work by choosing original test suites adequate for a variety of coverage criteria.

Threats to construct validity are factors in the experiment design that may cause us to inadequately measure concepts of interest. In our experiments, we made several simplifying assumptions in the area of costs. In test suite reduction, we are primarily interested in two different effects on costs. First, there is the cost savings obtained by running fewer test cases. In this study, we assume that each test case has a uniform cost of running (processor time) and monitoring (human time). These assumptions may not hold in practice. The second cost of interest is the cost of failing to find faults during testing as a result of running fewer test cases. Here we assume that each fault contributes uniformly to the overall cost, which again may not hold in practice. These assumptions are commonly made in other studies of test suite reduction [[16][19]].

5.5. Data Collection Step

Using the JavaGUIReplayer application [14], we executed each test case in each test pool against the fault-free versions of the subject programs, collecting the unique call stacks generated by each test case. We repeated this process for line (statement) coverage using jcoverage [6] as our instrumentation tool. Method coverage was derived from the call stack coverage data. Because our tests were event-based, we knew their event coverage a priori. Our coverage statistics aggregated over the entire test pool for each application appear in Table 2.

Application	**Includes Library Data?**	**Terp Paint (TP)**	**Terp Word (TW)**	**Terp Spreadsheet (TS)**
# Call Stacks Observed	Yes	413166	569933	333882
# Methods Observed	Yes	12277	12665	11103
# Events	N/A	181	219	110
# Lines[1]	No	11803	9917	5381
# Classes[1]	No	330	197	135
# Methods[1]	No	1253	1380	746

Table 2: TerpOffice Static and Dynamic Program Elements

As noted earlier, our instrumentation process for call stack coverage incorporates the induced coverage of the supporting Java libraries. Because we used our raw call stack coverage data as the basis for method coverage, our method coverage approach also includes Java framework methods. However, it was not feasible

[1] Of TerpOffice source, as determined by jcoverage instrumentation.

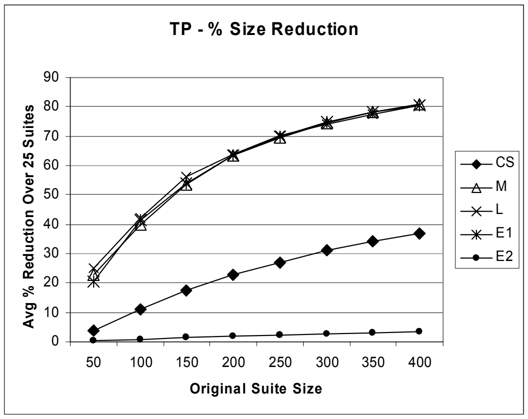
to instrument the entire Java SDK for line coverage, so our line coverage data is based solely on the TerpOffice source. Because of this, between the two approaches M and L, it is possible (and in fact the case) that we may cover more methods than lines.

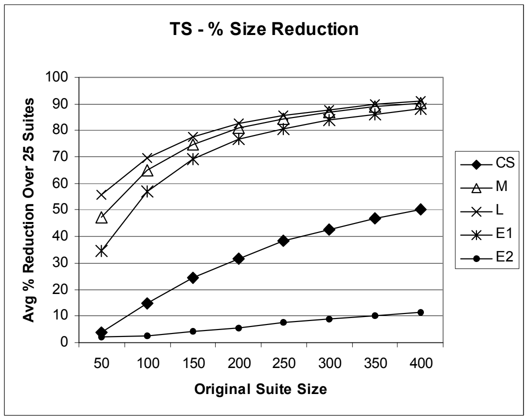
The data gathered during this step allowed us to create any number of test suites composed of the previously executed test cases and know the set of unique coverage elements and faults detected by the suite with no further execution of the program. Hence, it was not necessary to run each test suite under study against each version of the subject program. This simulation approach is similar to one used in [3] to evaluate adequacy criteria and test effectiveness.

5.6. Reduction Approach

Before reducing a test suite, we use the individual test case coverage information from step 5.2 to calculate the full set of unique call stacks that an execution of the full suite can be expected to generate. The full set is computed by *merging* the unique maximum-depth call stacks observed by each test case in the suite.

Here we must consider the situation where a maximum-depth call stack from one test case is not maximum-depth in another. For example, Test Case 1 (tc1) may generate the call stack c1 = <f1, f2, f3>, and Test Case 2 (tc2) may generate c2 = <f1, f2>. The call stack c2 is not maximum-depth in a test suite containing both tc1 and tc2. In our prior work [9], we addressed this issue by computing substack relationships between each pair of unique maximum-depth call stacks across the suite. In the example, this would lead to a selection of just tc1, because it covers both stacks c1 and c2. However, computing the substack relationships across an entire test suite with hundreds of thousands of unique (and deep) call stacks is very computationally expensive. Therefore, in this paper, we adopt a different approach, which is to forgo the computation of substack relationships and consider uniqueness of maximum-depth call stacks on a per-test-case basis. This approach is analogous to how we treat maximum-depth stacks across threads as discussed in Section 3.1. So in the example, this would lead to the inclusion of both tc1 and tc2. The consequence of this decision is that we forgo some potential size reduction in exchange for better runtime performance of the reduction process. Future work may quantify the delta in size reduction in practice.





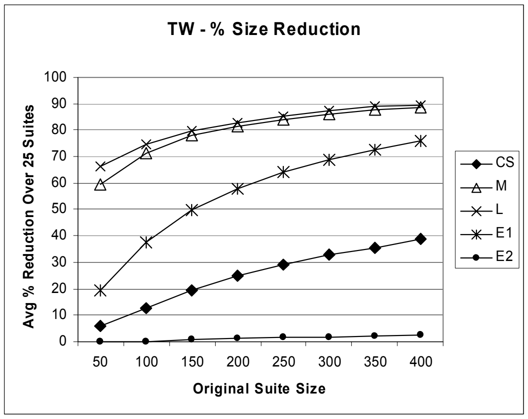


Figure 3: Percentage Size Reduction

After merging the unique maximum-depth call stacks from each test case in a given test suite, we apply the *ReduceTestSuite* heuristic [5] to compute the reduced test suite. Finally, we evaluate the size and fault detection capability of the reduced suite.

5.7. Experiment 1: Comparing Coverage-Based Reduction

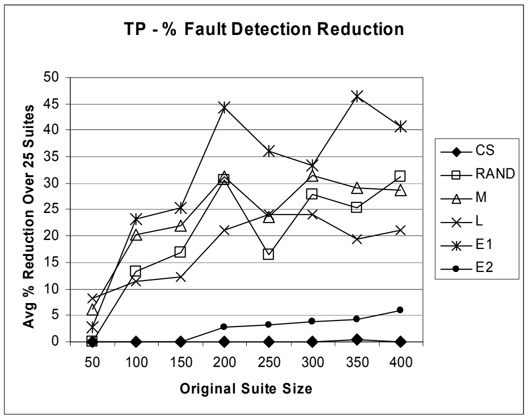
The goal of our first experiment was to reduce randomly generated test suites of various sizes based on call stack coverage (CS) and four other coverage criteria: event (E1), event-pairs (E2), line (L), and method (M). From the test universe, we evaluated suites ranging in size from 50 to 400, with 25 suites of each size. We reduced based on each of the five criteria and compared the percentage size reduction and percentage fault detection reduction metrics.

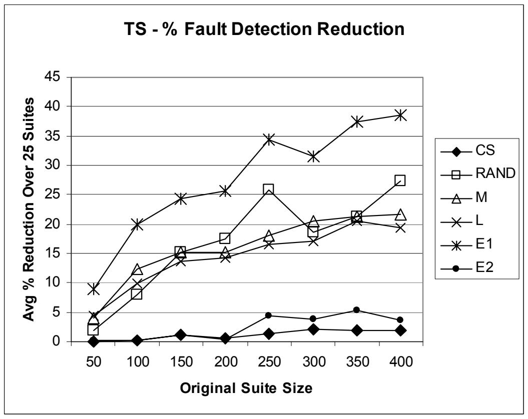
Size Reduction

Percentage size reduction results for the three applications TP, TS, and TW appear in *Figure 3*. We see similar behavior in suite size reduction for all three applications. E2 displays very little size reduction in all cases, which is expected because the original test cases were generated using an algorithm based on event flow. E1, M, and L are very close except in TW, where E1 results in less size reduction than M and L (but still notably more than CS). The CS technique strikes a middle ground between E2 (and no reduction) and the other three techniques, yielding 38-50% reduction for the largest suite size.

Fault Detection Reduction

Percentage fault detection reduction results for TP, TS, and TW appear in *Figure 4*. (The RAND technique will be discussed with Experiment 2 below.) The graphs are jagged due to the relatively small-magnitude and discrete nature of the fault data and the high sensitivity to the selection of specific test cases that may detect multiple faults. Nonetheless, some trends are clearly visible. As with percentage size reduction, there is no clear difference between M and L (recalling again that M includes methods from libraries and L does not). But call stack-based reduction is clearly favored over M, L, and E1, losing fault detection effectiveness in the 0-5% range for all applications and original suite sizes. Indeed, CS performs comparably to E2 even though E2-based reduction yields almost no size reduction in our empirical scenario. By comparison, using the traditional (non-GUI) *space* application as our test subject in [9], we observed percent fault detection reduction in the 12-16% range using both edge-coverage-adequate and randomly generated original suites. Clearly more subject applications need to be studied in future work, but this result suggests that call stack coverage analysis may be particularly applicable to GUI applications.





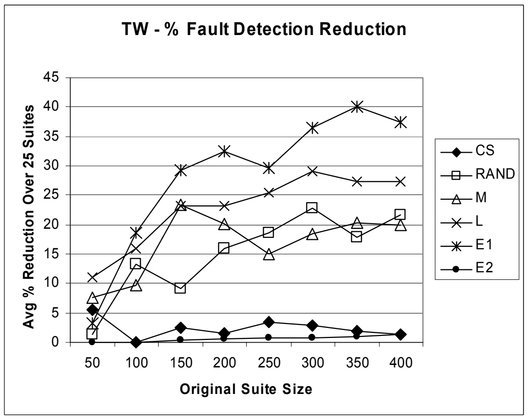


Figure 4: Percentage Fault Detection Reduction

5.8. Experiment 2: Controlling for Size of Reduced Suites

Our first experiment showed that call stack coverage excelled at preserving the fault detection effectiveness of reduced test suites. However, call stack-reduced suites were substantially larger than suites reduced by other criteria except for length-2 event sequences. Thus, it seemed possible that call stack coverage may have been preserving more fault detection solely on the basis of including more test cases. The goal of our second experiment was to evaluate this hypothesis.

Keeping the call stack-reduced suites from Experiment 1, we paired them with random suites of the same size (the RAND technique in *Figure 4*) and compared their fault detection effectiveness.

Referring back to Figure 4, RAND loses fault detection effectiveness comparable to L and M, thus performing considerably worse than CS. Considering that the RAND suite sizes are equal to those of CS, we conclude that call stack coverage contains valuable information that preserves fault detecting ability of test suites under reduction.

5.9. Discussion

In our experiments, call stack coverage-based reduction resulted in considerably larger reduced suite sizes than approaches based on method, line, or simple event flow coverage. In exchange for the larger reduced suite size, the call stack approach performed substantially better at retaining the fault detection capabilities of the original test suite. In practice, this may or may not be advantageous. For example, in a time-sensitive regression testing scenario, if there is sufficient time to run a call stack-reduced test suite in its entirety, our work suggests that it would be advisable to do so in order to obtain greater fault detection effectiveness. If time is more critical, a subset of the call stack reduced suite may be executed instead.

The feasibility of collecting call stack coverage in large multi-threaded and multi-language applications is a great benefit of the approach. However, where fault detection effectiveness is concerned, we believe that call stacks derive most of their power from their context sensitivity, capturing valuable information that most other coverage criteria miss. In future work, we will perform a missed-faults analysis across the techniques to quantify this conjecture.

6. Related Work

Rountev et al. [18] also consider the problem of "call chain" (call stack) coverage, beginning with a static analysis of potentially feasible call chains and dynamically measuring test coverage against it. They use the results of this analysis to guide the augmentation of a test suite to achieve higher coverage. Because the static analysis is conservative and therefore imprecise, achieving 100% coverage by these criteria is not in general possible. Unlike our work, they do not address the impact of this type of coverage on test suite reduction.

There have been numerous studies of test suite reduction and its relationship to fault detection effectiveness, including [[4], [16], [19]]. Jeffery and Gupta [7] present a test suite reduction approach that combines two different coverage criteria ("primary" and "secondary") to achieve improved reduced suite fault detection effectiveness with "selective redundancy". Call stack coverage would be an interesting choice as a participant in this technique, perhaps as a secondary participant with one of the simpler but context-insensitive criteria such as statement or branch coverage. Leon and Podgurski [8] apply clustering algorithms to the test suite reduction problem instead of the traditional coverage maximization approach. Again we feel that the context-preserving nature of call stack coverage would make it an excellent criterion on which to cluster test cases.

The Rostra framework [20] collects method sequences on a given object in an object-oriented system. The sequences are then used as coverage criteria for test suite reduction (among other applications). Unlike Rostra, our call stack technique is global and makes no assumptions about the threading behavior of test case executions.

7. Conclusions and Future Work

In this paper, we presented tools and techniques that allow us to dynamically collect call stacks in multithreaded GUI applications, including entries from the libraries that they use. And we empirically demonstrated the feasibility and effectiveness of using dynamically collected call stacks as a coverage criterion for GUI applications.

We have shown that event-driven GUI applications are sufficiently different from traditional applications to require new coverage criteria [12]. In future work, we plan to further generalize our results for coverage criteria that are effective for GUI testing scenarios.

Although we were able to successfully analyze complete call stack coverage data for the TerpOffice applications, the data volume for even larger applications may become unwieldy. Thus, we intend to look for techniques that reduce the number of coverage requirements generated by a complete call stack data collection while still retaining call stack coverage's desirable qualities. One idea is to limit the depth of calls into library routines. Another strategy is to define a "similarity metric" for call stacks such that different stacks with a certain similarity value may be considered redundant and therefore be discarded.

To further explore the notion that the context provided by call stacks is valuable in test suite reduction, we will perform a missed-faults analysis. By inspecting code related to faults found by call stack reduced suites but missed by other reduced suites, it may be possible to qualify the importance of calling context.

Finally, we believe there is a need to better quantify the tradeoffs between fault detection effectiveness reduction and size reduction. We will develop, apply, and evaluate new metrics to assist practitioners when considering test suite reduction approaches.

Acknowledgements

The authors thank the anonymous reviewers of this paper whose feedback and comments played an important role in reshaping the experimental results and analyses. This work was partially supported by the US National Science Foundation under NSF grant CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421.

9. References

[1] G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. SIGPLAN '97 *Conf. on Programming Language Design and Implementation*, 1997.

[2] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering Volume 28, no. 2*, February, 2002, pages 159-182.

[3] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. *ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering,* Nov. 1998.

[4] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. *Proceedings of the 25th International Conference on Software Engineering*, pp. 60-71, 2003, Porland, Oregon, United States.

[5] 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 (TOSEM)* July 1993 Volume 2 Issue 3.

[6] jcoverage information on the web at http://www.jcoverage.com/, April, 2006.

[7] D. Jeffrey and N. Gupta. Test suite reduction with selective redundancy. *IEEE International Conference on Software Maintenance (ICSM) 2005*, pages 549-558, Budapest, Hungary, 2005.

[8] D. Leon and A. Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003)*, November 2003, Denver, Colorado, United States.

[9] S. McMaster and A. Memon. Call stack coverage for test suite reduction. *IEEE International Conference on Software Maintenance (ICSM) 2005*, pages 539-548, Budapest, Hungary, 2005.

[10] A. Memon, A. Nagarajan, and Q. Xie. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(1):27.64, 2005.

[11] A. Memon, M. Pollack, and M. L. Soffa. Automated test oracles for GUIs. *SIGSOFT Eighth International Symposium on the Foundations of Software Engineering (2000),* pages 30-39, San Diego, California, USA, 2000.

[12] A. Memon, M. Pollack, M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering 27(2),* pages 144-155, (2001).

[13] A. Memon, M. L. Soffa, and M. Pollack. Coverage criteria for GUI testing. *ESEC / SIGSOFT FSE 2001*, pages 256-267, Vienna, Austria, 2001.

[14] A. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 884-896, October, 2005.

[15] S. Rapps. and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on. Software Engineering.* 11, 4 (Apr. 1985), 367-375.

[16] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong. Empirical studies of test-suite reduction. *Journal of Software Testing, Verification, and Reliability,* V. 12, no. 4, December, 2002.

[17] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *IEEE Transactions on Software Engineering,* vol. 27, no. 10, pp. 929-948, October, 2001.

[18] A. Rountev, S. Kagan, and M. Gibas, Static and dynamic analysis of call chains in Java. *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04)*, pages 1-11, July 2004.

[19] W. E. Wong, J. R. Horgan, S. London, A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Proceedings of the 17th International Conference on Software Engineering*, p.41-50, 1995, Seattle, Washington, United States.

[20] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit Tests. *19th IEEE International Conference on Automated Software Engineering*, Sep. 2004, pp. 196-205, Linz, Austria.