Call-Stack Coverage for GUI Test Suite Reduction

Scott McMaster, *Member*, *IEEE Computer Society*, and Atif M. Memon, *Member*, *IEEE*

Abstract—Graphical user interfaces (GUIs) are used as front ends to most of today's software 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 multilanguage GUI implementations, callbacks for event handlers, virtual function calls, reflection, and multithreading. Moreover, many existing techniques ignore code in libraries and fail to consider the context in which event handlers execute. Consequently, they yield GUI test suites with seriously impaired fault-detection abilities. This paper presents a reduction technique based on the **call-stack coverage criterion**. Call stacks may be collected for any executing program with very little overhead. Empirical studies in this paper compare reduction based on call-stack coverage to reduction based on line, method, and event coverage, including variations that control for the size and optional consideration of library methods. These studies show that call-stack-based reduction provides unique trade-offs between the reduction in test suite size and the loss of fault detection effectiveness, which may be valuable in practice. Additionally, an analysis of the relationship between coverage requirements and fault-revealing test cases is presented.

Index Terms—Testing strategies, test coverage of code, test management, testing tools.

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 the overall application code and 2) due to the GUI's proximity to the 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 [19]. A recent test-case generation technique based on *event-flow coverage* has been shown to be effective for defect detection in GUI applications [20]. 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 [7], [31], [25] 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. 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 [25], data flow [31], or dynamic program invariants [6]. In such an approach, each *coverage requirement* (that is, for method coverage, each method) covered by the original full test suite is also covered by the resulting reduced test suite. Traditionally, these approaches have been developed for and evaluated against conventional non-GUI software.

We believe that GUI-intensive software poses new challenges for coverage-based testing that require the development of new solutions. More specifically, the execution model for a GUI, based on an event-listener loop, differs from that of other types of software. During GUI execution, users perform actions that result in events. In response, each event's corresponding event handler is executed. The order in which event handlers execute depends largely on the order in which the user initiates the events. Hence, in a GUI application, a given piece of code called via an event handler may be executed in many different contexts due to the increased degree 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, multithreading, and event-handler callbacks. These features severely impair the applicability of techniques that rely on static analysis or the availability of language-specific and/or format-specific instrumentation tools.

This paper extends previous work on test suite reduction based on the call-stack coverage criterion. A *call stack* is a sequence of active calls associated with each thread in a stack-based architecture. Methods are pushed onto the stack when they are called and are popped when they return or when an exception is thrown (where supported, as in Java or C++). An example of a call stack from the simple Java program in Fig. 1a appears in Fig. 1b. This call stack was collected by the tools that we developed and will be discussed in detail in Section 4. In Fig. 1b, each line contains a method parameter list, return type, and name, including any package or namespace qualifiers. At the bottom of the stack appear the program's entry point, `main`, and the

• *The authors are with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: {scottmcm, atif}@cs.umd.edu.*

Manuscript received 15 Nov. 2006; revised 5 July 2007; accepted 8 Oct. 2007; published online 16 Oct. 2007.
Recommended for acceptance by D. Hoffman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0257-1106.
Digital Object Identifier no. 10.1109/TSE.2007.70756.

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

(a)

```
(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
```

(b)

Fig. 1. (a) A Hello-World example. (b) Associated call stack.

`println` method call shown in Fig. 1a. Above them are a number of library methods invoked as a consequence of the call to `println`.

The basic intuition behind call stack-based reduction is that two test cases are "equivalent" if they generate the same set of call stacks; hence, one of them could be eliminated to conserve resources. Unlike criteria such as line or branch coverage, call stack coverage has the benefit of encapsulating valuable context information. Aside from having the advantage of taking into account the context in which a method is called and the relative ease with which call stacks may be collected, call stack-based reduction has the following additional advantages for modern GUI applications:

- *Libraries and frameworks.* These are essential to modern software development in general and GUI applications in particular. Many test coverage techniques only collect coverage requirement data based on the instrumentation of first-party application source or object code. The reasons for this include the unavailability of the 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 trade-off, coverage techniques potentially overlook vast amounts of interesting behavior induced in the library code by the application. For example, consider the program in Fig. 2. If no library code is instrumented, every execution of this program against an integral input will satisfy line, branch, and data-flow coverage. Thus, when used in test suite reduction, each of those coverage approaches could potentially drop all tests that exercise the code with the integral input greater than or less than zero, thereby missing the array index out of bounds exception that occurs with such an input. In contrast, the call stack coverage technique presented in this paper 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.
- *Object-oriented language features.* Modern GUI application frameworks, usually implemented in languages like C++, Java, and C#, make extensive use of object-oriented programming (OOP) 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, in all cases, the stack contains the actual methods invoked. Consider the program shown in Fig. 3, which takes two command-line arguments to the main method: 1) a method name presumed to be `toUpperCase` or `toLowerCase` and 2) a string argument to be passed to the specified method via a dynamic invocation using Java's reflection mechanism. Because of the use of reflection, the call stacks generated by various executions of this program will differ based on the method name parameter. Clearly, this is a behavior that should be captured for the purposes of test suite reduction. However, static analysis cannot, in general, determine that `toUpperCase` or `toLowerCase` may be invoked by this program. 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.
- *Multithreading.* Modern GUI applications are all multithreaded. Indeed, all *Java* applications are multithreaded, if for no other reason than the presence of the garbage collector. A reduction technique should take into account the impact of multiple threads on software errors. Call stack coverage can be efficiently collected and processed in a multithreaded environment.
- *Multilanguage implementations.* Unlike other traditional coverage criteria such as data-flow/def-use pairs [23], call stack coverage is easily captured in a

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

Fig. 2. A simple program demonstrating the impact of library code on errors.

```
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() );
       }
```

Fig. 3. A simple example demonstrating the impact of OOP features on errors.

multilanguage application, with or without the availability of the source code. In general, writing a tool for collecting call stacks only requires method entry and exit hooks, which already exist on most compilers or runtime platforms to enable the construction of call profilers. A large GUI application implemented in multiple languages is no different from a single-language implementation when abstracted via the runtime call stack.

A preliminary model of call stacks has already been developed and reported [15], [16]. Call stack coverage was shown to be a practical and effective basis for performing test suite reduction, advancing the state of the art for coverage-based test suite reduction. Empirical evaluation indicated that call stack-coverage-based test suite reduction produces better results for GUI applications compared to traditional techniques. The work is now further extended by comparing additional reduction approaches and proposing novel analyses for evaluating the performance of test suite reduction techniques. The new experiments and analyses presented in this paper further demonstrate the value of call stack-based test suite reduction and provide some justification for its effectiveness.

Contributions. This paper makes the following contributions to the field of test suite reduction and GUI testing:

1. It empirically evaluates call stacks as a coverage criterion for test suite reduction versus several traditional coverage criteria when holding reduced suite size constant.
2. It investigates the importance of including library and framework coverage information when reducing test suites.
3. It proposes a new single-point metric for the effectiveness of test suite reduction techniques.
4. It analyzes coverage-based test suite reduction techniques from the standpoint of how many faults may theoretically be missed in reduced suites drawn from a given universe.

Structure of the paper. The remainder of this paper is structured as follows: Section 2 presents related work. Section 3 provides a model and definition for working with call stacks in program analysis. Section 4 describes our implementation approach for collecting call stacks and utilizing them in test suite reduction. Section 5 discusses several test suite reduction experiments in detail and Section 6 presents the application of new analysis techniques to our experimental results. Section 7 concludes and proposes directions for future work.

2 Related Work

To the best of our knowledge, call stacks have not been used before as a criterion for coverage-based test suite reduction. Several researchers have presented ideas that are relevant to this research. Related research for the areas of GUI testing, test suite reduction, and call chains is presented here.

Test reduction. There have been numerous studies of test suite reduction and its relationship to fault detection effectiveness. Harder et al. [6] use dynamic invariant detection techniques to create a reduced test suite. While running a program, they maintain an "operational abstraction," which is a mathematical picture of the program's dynamic behavior. The "operational difference" technique applied to test suite reduction executes each test case in a suite in turn and, if a test case does not change the current operational abstraction of the program, it is discarded. Like call stack reduction (and unlike most other reduction techniques), this approach makes use of dynamic program behavior rather than syntax. However, it has significant performance overhead. Wong et al. [31] reduce relative to the all-uses coverage criterion and observe little or no fault detection effectiveness reduction in the reduced suites. They also find a direct relationship between the ease of finding faults and the likelihood that they will be detected after reduction. In contrast, Rothermel et al. [24] reduce with respect to all-edges coverage and find significant reductions in fault detection effectiveness. They contrast their results with those in [31] and suggest possible causes for the different conclusions. However, collecting all-uses and other

data-flow coverage information generally requires tools that may be difficult to build and use for certain environments, particularly against an application built using multiple programming languages [8]. In contrast, call stack coverage information is relatively simple to obtain by using tools that we have developed and made available [10]. Additionally, call stack coverage can be analyzed on any stack-based runtime environment, which encompasses most language and system combinations in practical use today.

Jeffery and Gupta [13] 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 [14] and Dickinson et al. [2] apply clustering algorithms to the test suite reduction problem instead of the traditional coverage maximization approach. Again, we believe that the context-preserving nature of call stack coverage would make it an excellent criterion on which to cluster test cases.

Sampath et al. use concept analysis to generate minimal test suites from user sessions defined as URLs in a Web application [28]. Their approach has the interesting property that test suites can be incrementally updated as new user session data becomes available. Although Web application URLs model program behavior at a very different level of abstraction from call stacks, it is possible that methods in a call stack could be arranged in a concept lattice and a similar reduction technique applied.

The test suite reduction problem is closely related to *test case prioritization* [4] because any reduction technique can be turned into a prioritization technique by repeated applications of the reduction algorithm to the remainder of the suite.

GUI testing. Our work is particularly concerned with developing new coverage criteria for GUI applications. Event-based coverage [20] is specially tailored for use in GUI applications for which test cases can be modeled as sequences of events. Events may be menu invocations, button clicks, key presses, etc. The experiments in Section 5 use two different event coverage criteria, called event (E1) and event interaction (E2). In E1, each event in isolation is a coverage requirement, while, in E2, unique pairs of events are included as coverage requirements. E1 and E2 have been referred to as "event coverage" and "event-interaction coverage," respectively, in [20].

Call chains. Rountev et al. [27] 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 percent coverage by these criteria is not, in general, possible. Unlike our work, the authors do not address the impact of this type of coverage on test suite reduction.

The Rostra framework [32] 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 operates on an entire program rather than individual objects. Our technique also makes no assumptions about the threading behavior of test case executions or the usage of shared variables.

3 Modeling and Collecting Call Stacks

There are multiple ways of modeling and representing call stacks for use in a test suite reduction process. In Fig. 1b, a call stack is represented by the full method signature of each active method. Other possible approaches include capturing each active method by its method name only or by a full signature plus parameter values. Additionally, each representation may be augmented by a maximum allowable depth of recursion. In practice, the chosen call stack representation will have an impact on the feasibility of the reduction technique. Some models may generate so many different call stacks that collection and analysis is infeasible from a resource perspective. Other methods may generate so few different call stacks that differences between test cases are lost and fault detection effectiveness is compromised as a result. Due to the 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, when using full method signatures, the simple program in Fig. 1a generated 803 call stacks. Our subject applications built with Java Swing generated hundreds of thousands.

Definitions. Each running thread in a multithreaded 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. 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:

$$c_s = < m_1, m_2, \ldots m_i >, i < n, \tag{1}$$

$$c^s = < m_1, m_2, \ldots m_n, \ldots m_i >, i > n. \tag{2}$$

Let the set of all unique stacks generated by a thread t be denoted as $C(t)$. 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 the deepest, or *maximum depth*, stacks $C(t)_{max}$ in a thread t as follows:

$$C(t)_{max} = \{C \in C(t) | C(t)^s = \emptyset\}, \tag{3}$$

where $\emptyset$ is the empty set. That is, $C(t)_{max}$ is the set of all call stacks that do not have any superstacks. Since each maximum depth stack implies the existence of all of its substacks in $C(t)$, $C(t)_{max}$ is a more compact representation of the set of all unique call stacks generated by thread t.

To characterize the behavior of an entire multithreaded 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:

$$T = < t_1, t_2, \ldots t_n > . \tag{4}$$

The set of unique call stacks for a program input I is

$$C_{max}(I) = \cup\{C(t)_{max} | t \in T\}. \quad (5)$$

$C_{max}(I)$ is the union of the sets of maximum-depth stacks observed on any thread and each element of $C_{max}(I)$ is a coverage requirement in our reduction technique. Note that the definition of $C_{max}(I)$ 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 $C_{max}(I)$. Therefore, $C_{max}(I)$ 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 $C(t)_{max}$ for each thread t is computationally very expensive and of marginal benefit.

We define a *test case* as input given to a program in order to test one or more aspects of the program. Running a test case tc from a test suite TS implies the execution of the program, which itself implies that a set of maximum depth call stacks $C_{max}(tc)$ generated by the execution can be associated with tc. We consider two test cases, tc_1 and tc_2, to be equivalent if they generate identical sets of maximum depth call stacks:

$$tc_1 \sim tc_2 \quad iff \quad C_{max}(tc_1) = C_{max}(tc_2). \quad (6)$$

Since a *test suite* is a set of test cases, we denote the union of all C_{max}s for all of the test cases in a test suite TS as

$$Stacks(TS) = \cup\{C_{max}(tc) \,|\, tc \in TS\}. \quad (7)$$

We define a *test suite reduction technique* to be an end-to-end approach for reducing the size of a test suite. For coverage-based test suite reduction, a technique consists of a coverage criterion and an algorithm for reducing the suite while holding the coverage of that criterion constant. Our proposed technique considers a maximum-depth call stack to be a *coverage requirement* in the test suite reduction algorithm *ReduceTestSuite* [7]. Thus, the execution of a reduced test suite $TS^{reduced}$ will generate the same set of unique call stacks as the execution of its original (full) counterpart TS^{full}, that is, $Stacks(TS^{full}) = Stacks(TS^{reduced})$.

An efficient data structure for recording call stacks on a given thread of execution is the *calling context tree* (CCT) [1]. The CCT is a tree data structure where the root represents the method that is the entry point of a thread and each child node represents a call to a specific method made by its parent. It is possible to construct a CCT efficiently at runtime by using the following process, which is discussed in more detail in Ammons et al. [1]:

1. Create a node representing the entry point of the thread and make it the current node.
2. When a method is called, do the following:
   a. If the current node has a child node representing the called method, make that the current node.
   b. If a node representing the called method is an ancestor of the current node, the call is recursive. Create a *backedge* to that ancestor node and make it the current node.
   c. If the current node does not have a child node representing the called method, create such a node and make it the current node.
3. When a method returns, set the current node to its parent.

While generally large for nontrivial applications, the size of the CCT data structure does not grow unbounded (as a full method trace would) over the runtime of a test case, thus making the resulting data volume constant and manageable. Once a CCT is constructed, the set of unique maximum-depth call stacks recorded in that CCT may be calculated by traversing each path to a leaf in the tree.

4 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. When a thread exits, its CCT is traversed to calculate the set of unique call stacks seen on that thread 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.

To illustrate our technique's ability to work without source-level instrumentation, we built a Java Virtual Machine Tool Interface (JVMTI) agent, that is, JavaCCTAgent, to collect the CCT data necessary for a call stack coverage analysis for Java programs [10]. We chose to represent call stacks as an ordered set of full method signatures of the active methods, with at most one recursive invocation represented. For the Java-targeted implementation, we made use of the JVMTI hooks for method entry and method exit to maintain a CCT for each thread. Direct 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 ultimately written to the file system. We have built a similar tool for C/C++-based Windows applications using Detours [9] to instrument function entry and exit points. For the rest of this discussion, we focus on the JVMTI agent since it is the more advanced between the two implementations and is the implementation used in our primary experiments.

Since coverage is collected 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 nondeterminism into our data collection and, as a consequence, has an impact on the specific tests selected 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 the JVMTI agent consists of two files. The first file represents the observed call stacks as a list of tab-delimited method identifiers. The agent stores Java Native Interface (JNI) [11] 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. Thus, the 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, maps are used to create a canonical form based on the method signatures.

4.1 Reducing Test Suites

We use a C# implementation of the *ReduceTestSuite* algorithm presented in [7]. Because finding a minimal test suite that satisfies each coverage requirement is an NP-complete problem [7], *ReduceTestSuite* takes a heuristic approach. The algorithm includes in the reduced suite all test cases that cover a single coverage requirement. Then, it picks a test case that covers the most coverage requirements 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 requirements are covered. If n is the number of coverage requirements and m is the number of test cases, then the runtime of this algorithm is $\mathrm{O}(n^*\mathrm{Max}(m,n))$.

A different approach to coverage-based test suite reduction, known as the "ping-pong" heuristics, is given by Offutt et al. [22]. Using the "ping-pong" heuristics in call stack-based reduction is a possible avenue of future work.

5 Experiments

We implemented the call stack collection and reduction algorithms and performed five experiments to evaluate call stack-based test suite reduction.

5.1 Research Questions

This paper addresses the following research questions:

Q1. *How do the size and fault detection effectiveness of call stack-based reduced test suites compare to those of suites reduced on the basis of existing coverage criteria?*

Specifically, we wanted to directly compare the call stack-based technique (CS) to reduction based on four different types of coverage: event (E1), event-interaction (E2), line (L), and method (M) while using event-driven GUI applications.

Q2. *How does the fault detection effectiveness of call stack-based reduced test suites compare to suites of the same size created using other approaches?*

In the investigation of Q1, it is possible that reduced suites created using a given technique have better fault detection effectiveness due solely to the fact that the technique selects more test cases on the average than another technique. Q2 therefore removes size as an independent variable. Here, we wanted to investigate whether test suites created by call stack reduction preserved more fault-detecting ability than randomly reduced suites of the same size, as well as line, event, and method-reduced suites augmented with additional random test cases to make them the same size.

Q3. *How does including coverage information from third-party libraries affect the size and fault detection effectiveness of reduced test suites?*

Additionally, we wanted to evaluate the impact of including library routines in the method and call stack reduction on the size reduction and fault detection reduction.

Q4. *Does call stack-based test suite reduction perform differently in conventional and event-driven applications?*

Next, to see if call stack-based test suite reduction is sensitive to the type of application, we wanted to compare its behavior on a non-event-driven non-GUI application to what was observed for event-driven GUI applications.

Q5. *Are certain types of coverage requirements more likely to be associated with faults?*

If a specific coverage requirement is covered *only* by fault-revealing test cases, this intuitively provides strong evidence that the coverage requirement in question is related to a fault. Moreover, no coverage-preserving test suite reduction technique can possibly lose any such faults, which is very valuable to know. Thus, in practice, we might want to identify and select a coverage technique that maximizes the number of such coverage requirements. Therefore, we wanted to determine if coverage criteria differ in how strongly their coverage requirements are associated with faults.

To answer these research questions, we designed five experiments that we present next. In Experiment 1, we compared call stack-based reduction with event, event-interaction, line, and method-based reduction. Experiment 2 compared call stack reduction to randomly selected and augmented line, event, and method-reduced suites of the same size. In Experiment 3, we considered method and call stack coverage, excluding information about library methods. In Experiment 4, we compared call stack-based reduction with edge and method (function)-based reduction for a conventional application and Experiment 5 relates coverage requirements to fault-revealing test cases for various types of coverage.

5.2 Subject Applications

We used three applications from the TerpOffice Suite [21] as experimental subjects for Experiments 1-3.[1] TerpOffice is a business productivity suite written in Java by senior software engineering students over a period of years. The three applications under study are TerpPaint (TP), TerpWord (TW), and TerpSpreadsheet (TS). For Experiment 4, we used the well-studied *space* application [26], an antenna-steering system developed by the European Space Agency written in C and composed of about 6,200 noncommentary lines of code. Table 1 shows the key metrics for these applications' test suites. Each TerpOffice application is associated with a large universe of test cases generated using the event flow criterion [17]. Each application comes

1. Subject applications and other data are located at http://www.cs.umd.edu/~atif/Benchmarks/UMD2007a.html or contact the authors for more information.

TABLE 1
Test Cases and Faults

Application	TerpPaint (TP)	TerpWord (TW)	TerpSpreadsheet (TS)	Space (S)
Test Universe Size	1500	1000	1000	13585
# Detectable Faults (Versions)	43	18	101	34

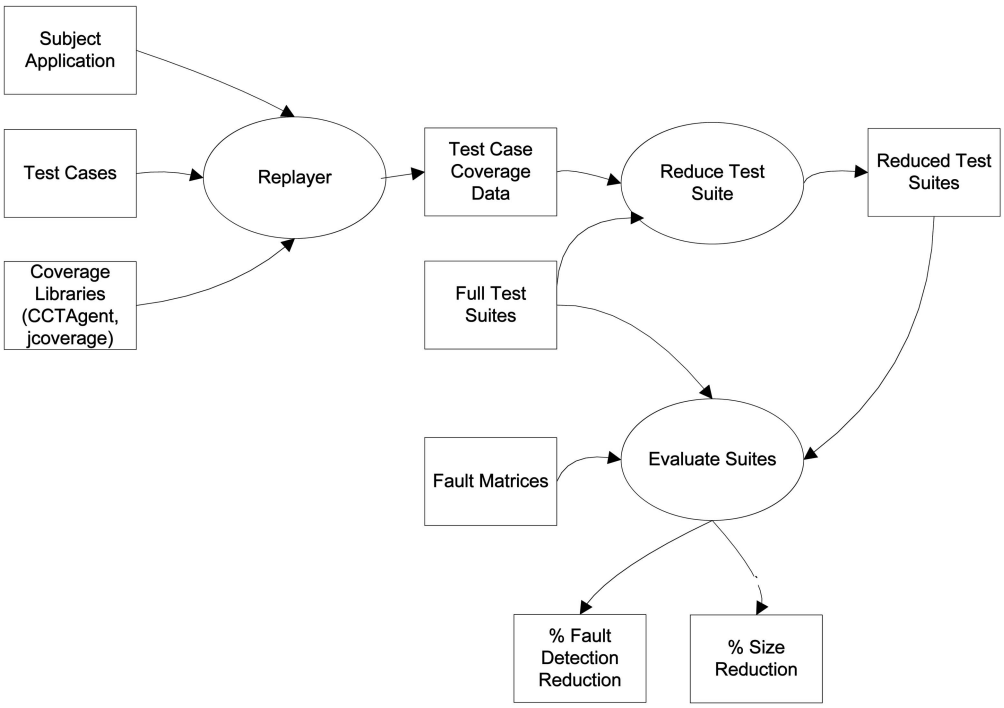

Fig. 4. Experimentation procedure.

with a set of single-fault versions, a set of known faults, and fault detection information for each test case.

5.3 Measured Variables

In this paper, fault detection effectiveness is measured on a per-test-suite basis, that is, 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 [24], [31]. For each reduction experiment, the captured metrics are the percent size reduction

$$100 * (1 - \text{Size}_{\text{Reduced}}/\text{Size}_{\text{Full}}) \quad (8)$$

and the percent fault detection reduction

$$100 * (1 - \text{FaultsDetected}_{\text{Reduced}}/\text{FaultsDetected}_{\text{Full}}). \quad (9)$$

Since these experiments deal with a fairly small number of discrete faults, averages of these quantities were taken over a large number of suites.

5.4 Experimentation Procedure

The experimentation procedure appears in Fig. 4. Ovals represent tools/processes; boxes represent experimentation artifacts/results. For each subject application, we begin with a pool of test cases, a set of known faults, and a fault matrix, that is, which test cases detect which faults. We then perform the following steps:

1. Randomly generate a set of test suites composed of test cases from the pool (not coverage-adequate for any particular criterion).
2. For each full (nonreduced) test suite, calculate the set of faults that it detects.
3. Select a coverage criterion.
4. Reduce each test suite while maintaining coverage relative to the selected criterion.
5. For each reduced test suite, calculate the set of faults that it detects.
6. Compute the percentage size reduction and percentage fault detection reduction.

We discuss this approach in more detail in the following sections.

5.5 Threats to Validity

Threats to external validity are factors that may impact our ability to generalize our results to other situations. The main threat to external validity in this study is the small sample size. This study collects test suite reduction data for only four programs, which were chosen for their availability. Three of these programs were constructed in a similar manner and may not be representative of the broader

TABLE 2
TerpOffice Static and Dynamic Program Elements

	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[2]	No	11803	9917	5381
# Classes[2]	No	330	197	135
# Methods[2]	No	1253	1380	746

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

population of programs. An experiment that would be more readily generalized would include multiple programs of different sizes and from different domains. Additionally, one would expect the effectiveness of the call stack reduction process to vary depending on the 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. Three of the subject applications used in this paper are GUI-event-driven and thus contain many small event-handling methods. This should increase the effectiveness of the call stack-based reduction technique relative to what it could do against an application that implemented the same behavior using relatively fewer or more monolithic functions as we see in *space*. (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, the characteristics of the 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 the 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 these experiments, several simplifying assumptions were made in the area of costs. In test suite reduction, researchers 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, it is assumed 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 [25], [31]. Because test suite reduction seeks to permanently reduce the size of a test suite by discarding redundant or less effective test cases, the cost of applying a given reduction technique is amortized across all future executions of the test suite and is therefore not factored into these experiments.

Finally, for feasibility reasons, line coverage data for TerpOffice did not include coverage of the underlying library code, in contrast to the approach taken for the method coverage. Including the line coverage of libraries may alter the performance of line-based test suite reduction relative to the other coverage criteria.

Threats to internal validity include the possibility of defects in the tools used in the experiments and errors in the execution of the experimental procedure, any of which may impact the accuracy of our results and the conclusions that we draw from them. These threats have been controlled for through testing the tools and the data quality.

5.6 Data Collection Step

In Experiments 1-3, using the JavaGUIReplayer application [21], shown as "Replayer" in Fig. 4, each test case in each test pool was executed against the fault-free versions of the subject programs, collecting the unique call stacks generated by each test case. This process was repeated for the line coverage by using jcoverage [12] as the instrumentation tool. The method coverage was derived from the call stack coverage data. Because the tests in Experiments 1-3 were event-based, their event coverage was known a priori. Coverage statistics aggregated over the entire test pool for each application appear in Table 2. For each subject application, the first two rows of Table 2 list the total number of unique call stacks and methods (including library methods, not limited to the TerpOffice source code) observed in a test run of the entire test universe. The next row shows the number of GUI events utilized in each application. Finally, the last three rows are static counts of lines, classes, and methods that comprise each application.

As noted in Section 4, our instrumentation process for call stack coverage incorporates the coverage of the supporting Java libraries induced by test case execution. Because we used the raw call stack coverage data as the basis for the method coverage, our method coverage approach also includes Java framework methods. However, because it was not feasible to instrument the entire Java SDK for the line coverage, 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 is, in fact, the case) that our tests 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 to know the set of unique coverage requirements 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 the one used by Frankl and Iakounenko [5] to evaluate adequacy criteria and test effectiveness.

5.7 Reduction Approach

Before reducing a test suite, the individual test case coverage information from Section 5.6 is used 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 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 a maximum depth in another. For example, test case 1 $(tc1)$ may generate the call stack $c1 = < m1, m2, m3 >$ and test case 2 $(tc2)$ may generate $c2 = < m1, m2 >$. The call stack $c2$ is not a maximum depth stack in a test suite containing both $tc1$ and $tc2$. In prior work [16] and in Experiment 4, this issue was addressed 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, as we observe in the TerpOffice applications, is very computationally expensive. Therefore, the experiments in this paper take a different approach, which is to forgo the computation of substack relationships and consider the uniqueness of maximum-depth call stacks on a per-test-case basis. This approach is analogous to how maximum-depth stacks are treated across threads, as discussed in Section 3.1. Thus, in the example, the reduction of a full test suite composed of both $tc1$ and $tc2$ would lead to the inclusion of both test cases in the reduced suite. The consequence of this decision is that this approach forgoes some potential size reduction in exchange for a better runtime performance of the reduction process. Future work may quantify the delta in size reduction in practice.

After merging the unique maximum-depth call stacks from each test case in a given test suite, we apply the *ReduceTestSuite* heuristic [7] to compute the reduced test suite. Finally, we evaluate the size and fault detection capability of the reduced suite.

5.8 Experiment 1: Comparing Coverage-Based Reduction

The goal of Experiment 1 was to reduce randomly generated test suites of various sizes based on the call-stack coverage and the four other coverage criteria: E1, E2, L, and M. From the test universe, suites ranging in size from 50 to 400, with 25 suites of each size, were evaluated. Test suites were reduced based on each of the five criteria and were compared in terms of the percent size reduction and the percent fault detection reduction metrics.

5.8.1 Size Reduction

Percentage size reduction results for the three applications TP, TS, and TW for each reduction approach appear in Fig. 5. (The SM and SCS approaches will be discussed in Section 5.10.) Similar behavior in suite size reduction is observed 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-reduced suites are smaller than M and L but are still notably larger than CS. The CS technique strikes a middle ground between E2 (and no reduction) and the other three techniques, yielding 38 percent to 50 percent of reduction for the largest suite size.

To evaluate the statistical significance of differences between CS and the other techniques, paired-t testing was performed at the 0.05 level, with the null hypothesis that there is no statistically significant difference between the means of "CS percent size reduction" and the means of each of the other techniques. The results appear in the left half of Table 3. (Reduction techniques LA, MA, and E1A will be defined in Experiment 2, Section 5.9.) Since all of the p-values for the percentage of size reduction are below 0.05, the null hypothesis is rejected and the alternative hypothesis, which is that there is a statistically significant difference between the means of CS and the other techniques, is accepted.

5.8.2 Fault Detection Reduction

The percentage of fault detection reduction results for TP, TS, and TW appear in Fig. 6. (The RAND, E1A, LA, MA, SCS, and SM techniques will be discussed in subsequent experiments.) 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 the percentage of size reduction, there is no clear difference between M and L (recalling again that M includes methods from libraries and L does not). However, call stack-based reduction is clearly favored over M, L, and E1, losing fault detection effectiveness in the 0 percent to 5 percent 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 experiments. By comparison, using the traditional (non-GUI) *space* application as the test subject in our previous work [16], we observed percent fault detection reduction in the 12 percent to 16 percent 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.

To evaluate the statistical significance of the difference of means between CS and M, L, E1, and E2, respectively, paired-t testing was performed at the 0.05 level, with the null hypothesis that there is no statistically significant difference between "CS fault detection reduction" to each of the other techniques. The results appear in the right half of Table 3. Since all p-values of M, L, E1, and E2 for the percentage of fault detection reduction are below 0.05, the null hypothesis is rejected and the alternative hypothesis, which is that there is a statistically significant difference between the means of CS and other techniques for all subject applications, is accepted.

In summary, we find that CS-based reduction on test suites for event-driven applications results in measurable size reduction and extremely low fault detection reduction compared to other techniques. This result answers Q1.

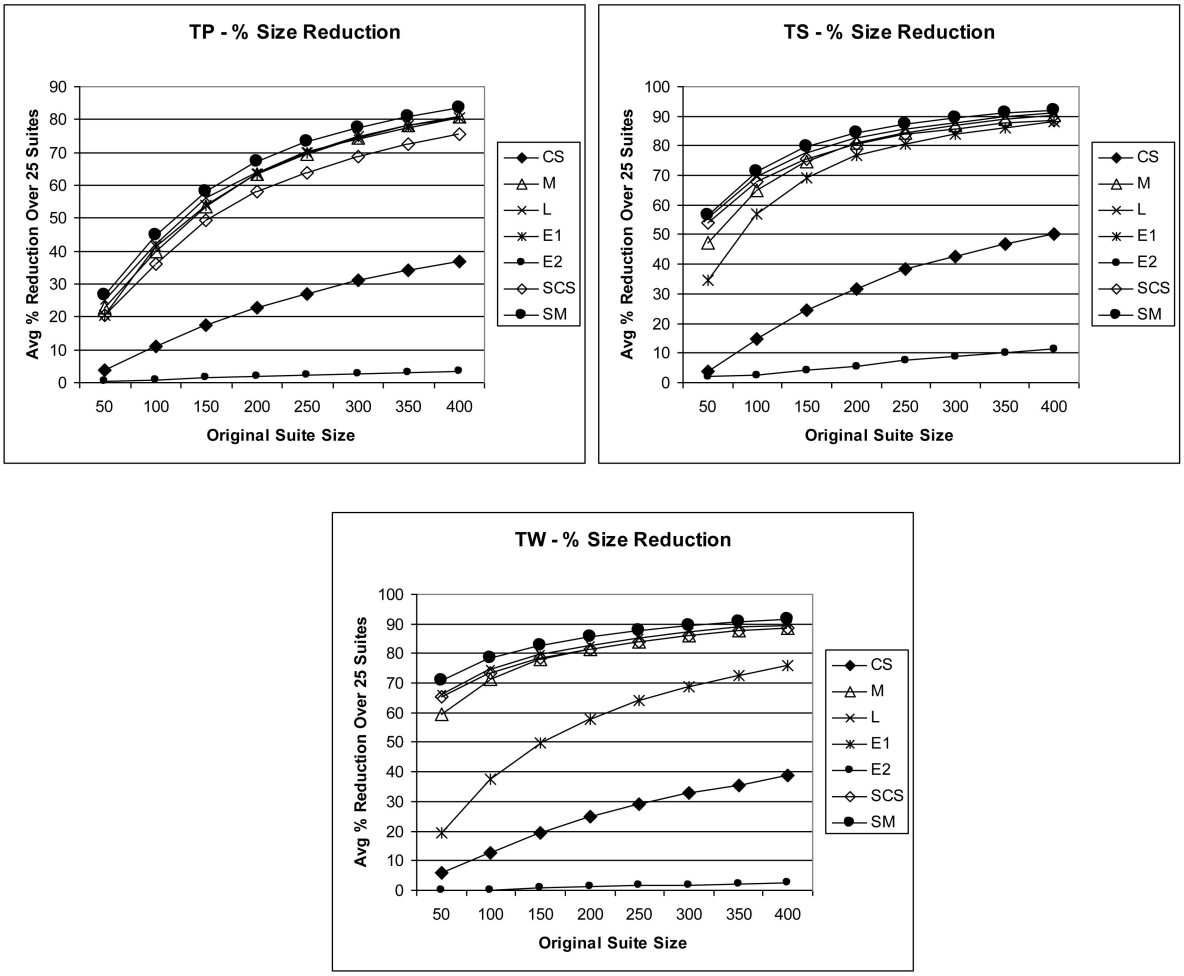

Fig. 5. Percentage size reduction.

TABLE 3
Paired-t Testing of CS versus Other Techniques (Bold Values Not Statistically Significant at the 0.05 Level)

CS vs.	% Size Reduction p-Value			% Fault Detection Reduction p-Value		
	TP	TS	TW	TP	TS	TW
RAND	--	--	--	0.001041	0.000803	0.002916
M	7.84E-06	6.04E-09	3.52E-10	8.48E-05	8.13E-05	0.000353
L	3.02E-06	2.9E-08	1.29E-09	8.07E-05	7.02E-05	7.26E-05
E1	1.13E-05	4.59E-08	1.36E-05	0.000426	8.9E-05	0.000792
E2	0.000823	0.000932	0.000414	0.016876	0.039215	0.025051
LA	--	--	--	0.007803	0.002918	***0.553965***
MA	--	--	--	0.006307	0.002236	***0.10448***
E1A	--	--	--	0.000976	0.005401	0.010153
SCS	7.13E-06	4.85E-08	1.54E-09	4.63E-05	1.11E-07	3.89E-05
SM	2.95E-06	2.96E-08	3.61E-09	4.78E-05	4.68E-05	4.35E-05

5.9 Experiment 2: Controlling for Size of Reduced Suites

Experiment 1 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 E2. Thus, it seemed possible that the call stack coverage may have been preserving more fault detection solely on the basis of including more test cases. The goal of Experiment 2 was to evaluate this hypothesis. The call stack-reduced suites from Experiment 1 were paired with random suites of the same size (the RAND technique in Fig. 6) and compared with respect to their fault detection effectiveness. We also took the reduced suites resulting from L, M, and E1 and randomly augmented them with additional test cases drawn from the full test suites so that the augmented suite sizes were equal to the CS suite sizes derived from each full test suite. These "additional"

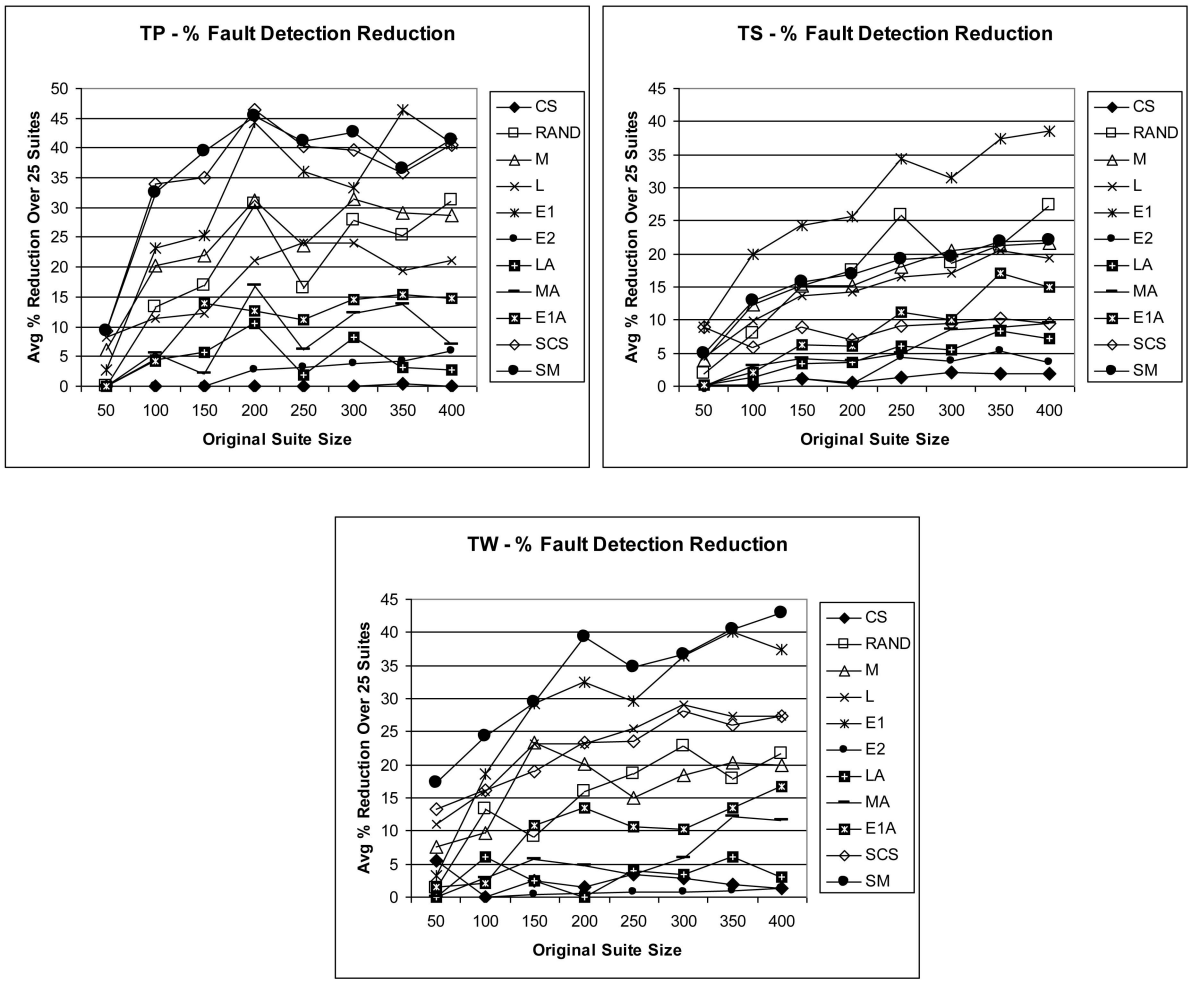

Fig. 6. Percentage fault detection reduction.

TABLE 4
Non-Library Coverage Statistics

Application	Observed Method Count Excluding Methods	Observed Call Stack Count Excluding Library Methods
TerpPaint	680	923
TerpWord	757	1780
TerpSpreadsheet	525	2653

techniques are the LA, MA, and E1A techniques, respectively, in Fig. 6.

Referring back to Fig. 6, RAND loses fault detection effectiveness comparable to the unaugmented L and M techniques, thus performing considerably worse than CS. The "additional" techniques perform better than RAND. According to Table 3, for two of the three subject applications, CS shows a significantly better percentage of fault detection reduction. For TW, the LA and MA techniques are not clearly better than CS. Considering that the suite sizes from RAND, E1A, LA, and MA are equal to those of CS, we conclude that, in most cases, call stack coverage contains valuable information that preserves the fault-detecting ability of test suites under reduction. This result addresses Q2.

5.10 Experiment 3: Omitting Library Methods

Most coverage techniques are evaluated only on those coverage requirements that can be derived from first-party source code. We have hypothesized that the ease with which the call stack coverage technique can incorporate context-sensitive coverage of library routines may be one of its major advantages.

To further explore this notion, we generated coverage information for both methods and call stacks, excluding methods from the Java platform libraries. These techniques are called SCS and SM in Figs. 5 and 6. The numbers of coverage requirements for the applications under study appear in Table 4. Because the TerpOffice applications highly leverage the Java platform libraries for their GUI and I/O support, omitting library methods from coverage results in far fewer coverage requirements.

TABLE 5
Paired-t Testing of SCS versus Other Techniques (Bold Values Not Statistically Significant at the 0.05 Level)

SCS vs.	% Size Reduction p-Value			% Fault Detection Reduction p-Value		
	TP	TS	TW	TP	TS	TW
M	1.79E-05	***0.520725***	***0.21264***	0.000252	0.006143	0.01079
L	1.75E-07	3.49E-08	2.95E-08	0.000322	0.009559	***0.305519***
E1	0.000321	0.032248	0.000448	***0.176047***	0.000728	0.047302
E2	7.11E-05	8.13E-08	8.42E-09	3.77E-05	3.27E-06	5.83E-06
SM	1.29E-07	3.75E-08	4.51E-06	***0.235898***	0.003313	0.000137

TABLE 6
Paired-t Testing of SM versus Other Techniques (Bold Values Not Statistically Significant at the 0.05 Level)

SM vs.	% Size Reduction p-Value			% Fault Detection Reduction p-Value		
	TP	TS	TW	TP	TS	TW
M	4.19E-06	0.002928	0.001513	0.000214	***0.052797***	8.68E-05
L	6.8E-06	1.54E-05	5.36E-05	0.000301	0.000156	0.000168
E1	4.11E-05	0.003248	0.000284	***0.125103***	0.000277	0.023723
E2	3.49E-05	7.24E-08	2.40E-09	3.96E-05	2.83E-05	1.19E-05
SCS	1.29E-07	3.75E-08	4.51E-06	***0.235898***	0.003313	0.000137

TABLE 7
Test-Suite Reduction for *space*

Means Over 1000 Test Suites							
Original		Edge-Reduced		Call Stack-Reduced		Random-Reduced	
Size	Faults Detected	Size	Faults Detected	Size	Faults Detected	Size	Faults Detected
2399.5	33.5	121.7	30.4	60.0	28.0	60.0	24.2
% Reduction From Original		90.1	9.2	95.2	16.3	95.2	27.6

When we perform test suite reduction based on the SCS and SM coverage data, size reduction is comparable to L, M, and E1 in all of the subject applications. Fault detection reduction displays quite a bit of variance between the applications. For TP, SCS and SM perform comparably to the least successful reduction technique E1. In TW, SM tracks again with E1, but SCS fares better and is comparable to the L-based technique. In TS, SM is similar to L, losing around 20 percent of its fault detection effectiveness for larger original suite sizes. However, SCS for TS does very well, losing not more than 10 percent of fault detection, which is significantly better than M, L, E1, E2, and SM, as shown in Table 5.

Looking back at Table 4, the success of the SCS technique seems to correlate with how many call stacks can be generated by an application's test suite, which itself can be highly influenced by the programming style. Specifically, an application written using many smaller methods (generally considered to be good OOP style) will generate more unique call stacks than an application written using larger more monolithic methods. Future work may explore this intuition in more detail.

Regardless, neither the SM nor the SCS technique approaches the CS technique at providing very small loss of fault detection. Results in Tables 3, 5, and 6 indicate statistically significant differences between both SM and M, and SCS and CS. Thus, we conclude that it is helpful to consider the coverage of library elements in a test suite reduction technique when the goal is to minimize the loss of fault detection effectiveness, answering Q3.

5.11 Experiment 4: Conventional Application

Experiment 4 seeks to determine how well the call stack coverage performs for non-event-driven (conventional) software and whether call stacks give us any insights into understanding the differences between conventional and event-driven software. To that end, the subject for Experiment 4 is the procedural C-language *space* application. *Space* is comprised of approximately 6,200 lines of code in 143 functions. The test suite for *space* generated 453 unique call stacks.

This experiment was made up of two parts. First, using 1,000 test suites for *space* used by Rothermel et al. [26], we reduced each test suite by using the call stack coverage (and compared our results to the results of Rothermel et al. for edge coverage [25]). As in Experiment 2, we also paired call stack-reduced suites with like-sized randomly reduced suites. The results appear in Table 7.

Second, randomly generated test suites of various sizes, 50 of each size between 50 and 1,000 for a total of 1,000 suites,

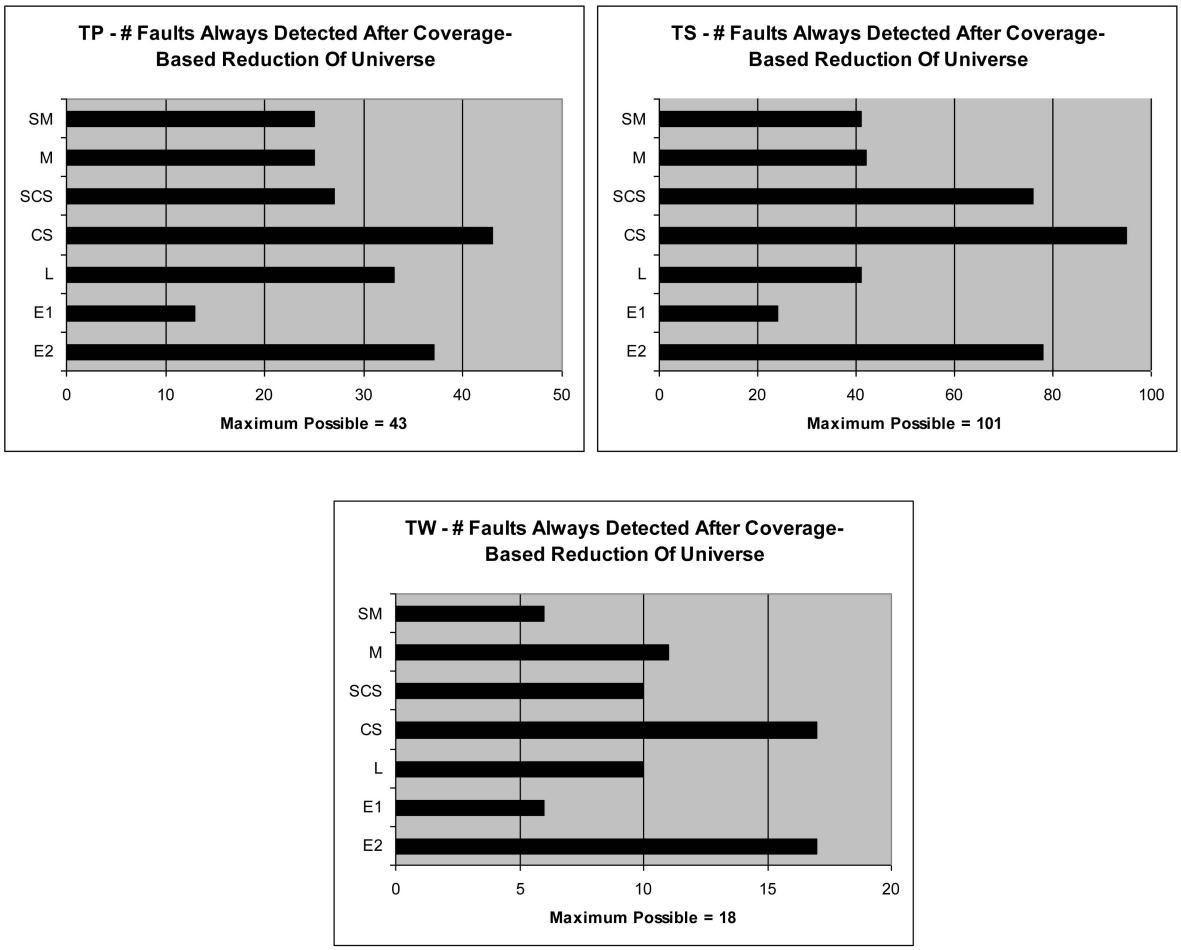

Fig. 7. Faults always found after reduction, by technique.

were evaluated. These suites were reduced using call stack coverage, function coverage, and random selection equal to the call stack reduced size. Over the randomly generated suites, the average fault detection reduction was 31.89 percent for function coverage, 24.54 percent for random reduction, and only 13.19 percent for call stack coverage.

Experiment 4 shows that call stack-based test suite reduction can provide a good trade-off between size reduction and fault detection reduction in a conventional application. However, compared to the findings in Experiments 1-3, call stack coverage seems to be a more effective criterion for test suite reduction against event-driven GUI applications than for conventional software. Although further research using a wider variety of GUI and conventional subject applications is needed, one possible explanation relates to the ability of call stacks to capture the context in which a given method is invoked. GUIs tend to have more degrees of freedom than conventional software. For example, the event-handling code for a particular event may execute differently, depending on the nature of the specific event invocation (that is, mouse versus keyboard) and the sequence of preceding events. Because each such scenario potentially results in a unique call stack, call stack-based test suite reduction will select such test cases and, as a consequence, their potentially unique fault-detecting capability. This result addresses Q4.

5.12 Experiment 5: Coverage Requirements and Fault-Revealing Test Cases

When using a test suite reduction technique that preserves the coverage of a given program element, a necessary condition for a fault to be missed by a reduced suite is that no coverage requirement is *only* covered by fault-revealing test cases. If one or more such coverage requirements exist, we intuitively expect an above-average probability that it is related in some way to the source of the fault. In this case, the reduction algorithm *must* select a fault-revealing test case; otherwise, coverage will be lost.

This observation led us to analyze our coverage and fault data to determine how many faults must be detected by any coverage-adequate reduced test suite on the entire test pool by using the various techniques CS, SCS, M, SM, L, E1, and E2. The results of this analysis appear in Fig. 7, where the x-axis shows the number of faults that will always be detected by any reduced suite, which is adequate for a given criterion on the y-axis.

The two method-based techniques, SM and M, and L perform similarly across applications. The context-sensitive SCS technique performs comparably in TP and TW and relatively better in TS. Looking at the call stack technique, in all but a small handful of cases, fault-revealing test cases generate call stacks that are never observed by non-fault-revealing test cases. This phenomenon provides an

explanation for the extremely low percentage of fault detection reduction observed for CS in Experiment 1, lending support to the hypothesis that context information enhances coverage-based test suite reduction and thus answering Q5. Further research is needed to characterize the non-CS techniques. Looking at the unique coverage requirement counts for individual faults, we see a number of cases where fault A is guaranteed to be detected by technique X but not by technique Y, but fault B for the same application is guaranteed to be detected by technique Y and not by technique X. This supports the idea that reduction approaches such as the one proposed by Jeffrey and Gupta [13], which incorporate two or more types of coverage data, may be more effective than using a single coverage criterion in isolation.

6 Analysis: Test Suite Reduction Metric

The feasibility of collecting call stack coverage in large multithreaded and multilanguage 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. Future work will include a missed-faults analysis across the techniques to quantify this conjecture.

In experiments using GUI applications as test subjects, 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 CS-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 or call stack reduction may be combined with another reduction criterion using an approach similar to the one proposed by Jeffrey and Gupta [13].

Prior work on test suite reduction provides very little guidance for practitioners who must make decisions about which reduction technique or techniques to use. If anything, the prior work emphasizes minimal fault detection reduction over size reduction. However, given trends in modern software development such as the increased use of test case generators and build-and-integration cycles often lasting a single day or less, this may not be the appropriate trade-off in practice. Because of this, there is a need for quantitative metrics and cost-benefit models that capture the size-versus-fault-detection trade-off to help guide practitioners make a more holistic choice when applying test suite reduction techniques.

In their work on test suite reduction in Web applications, Sampath et al. [29] propose a "figure of merit" (*fom*) for test suite reduction as

$$fom = redux * cvg * fd. \tag{10}$$

TABLE 8
Metric Weighting Scenarios

Scenario Number and Description	W_{SR}	W_{FDR}
1: Emphasize Small Suite Size	2.0	0.5
2: Emphasize Low Fault Detection Reduction	0.5	2.0
3: Equal Emphasis	1.0	1.0

Here, *redux* is the percentage of size reduction, fd is the percentage of faults *still detected* after reduction, and cvg is the percentage of coverage remaining for some specific criterion other than the one used in the reduction algorithm. This metric combines the desirability of high size reduction and the undesirability of high fault detection reduction into a single number.

A weakness of (10) is that the approach of using a simple product of terms does not allow practitioners to factor in the relative importance of size reduction and fault detection reduction when evaluating a technique. To solve this, we propose evaluating test suite reduction relative to the following single-point metric:

$$\begin{aligned} ReductionMetric = {} & (W_{SR} * \text{Percent } Size\ Reduction) \\ & + W_{FDR} * (100 - \text{Percent } Fault\ Detection\ Reduction). \end{aligned} \tag{11}$$

We define W_{SR} to be a weight representing the relative importance of size reduction in a given scenario. Similarly, W_{FDR} is a weight for the relative importance of fault detection reduction. We expect that practitioners will manipulate the weights to capture the relative importance of fault detection and size reduction in a specific scenario.

We consider three sets of weights defined in Table 8. In Scenario 1, a small reduced test suite size is deemed more important than low fault detection reduction. Scenario 2, conversely, considers low fault detection reduction the stronger factor. In Scenario 3, both measures are weighted equally. The selection of weights was made to keep the results from each scenario close in absolute magnitude. Conclusions should only be drawn based on relative values within a given scenario.

Applying the metric from (11) to the data collected in our experiments from Section 5 for the different reduction techniques, subject applications, and weighting scenarios yields the results in Fig. 8.

When a small suite size is the primary focus of the test suite reduction process (Scenario 1), the metric indicates that the favored techniques for GUI applications are based on the line coverage, method coverage including library methods, method coverage not including library methods (SM), and call stack coverage not including library methods (SCS). When low fault detection reduction is deemed more important (Scenario 2), the CS technique is preferred, followed closely by several other techniques with similar performance. With equal weighting applied to size reduction and fault detection reduction (Scenario 3), the relative metric values by technique again favor L, M, SM, and SCS, along with the improved performance of the "additional"

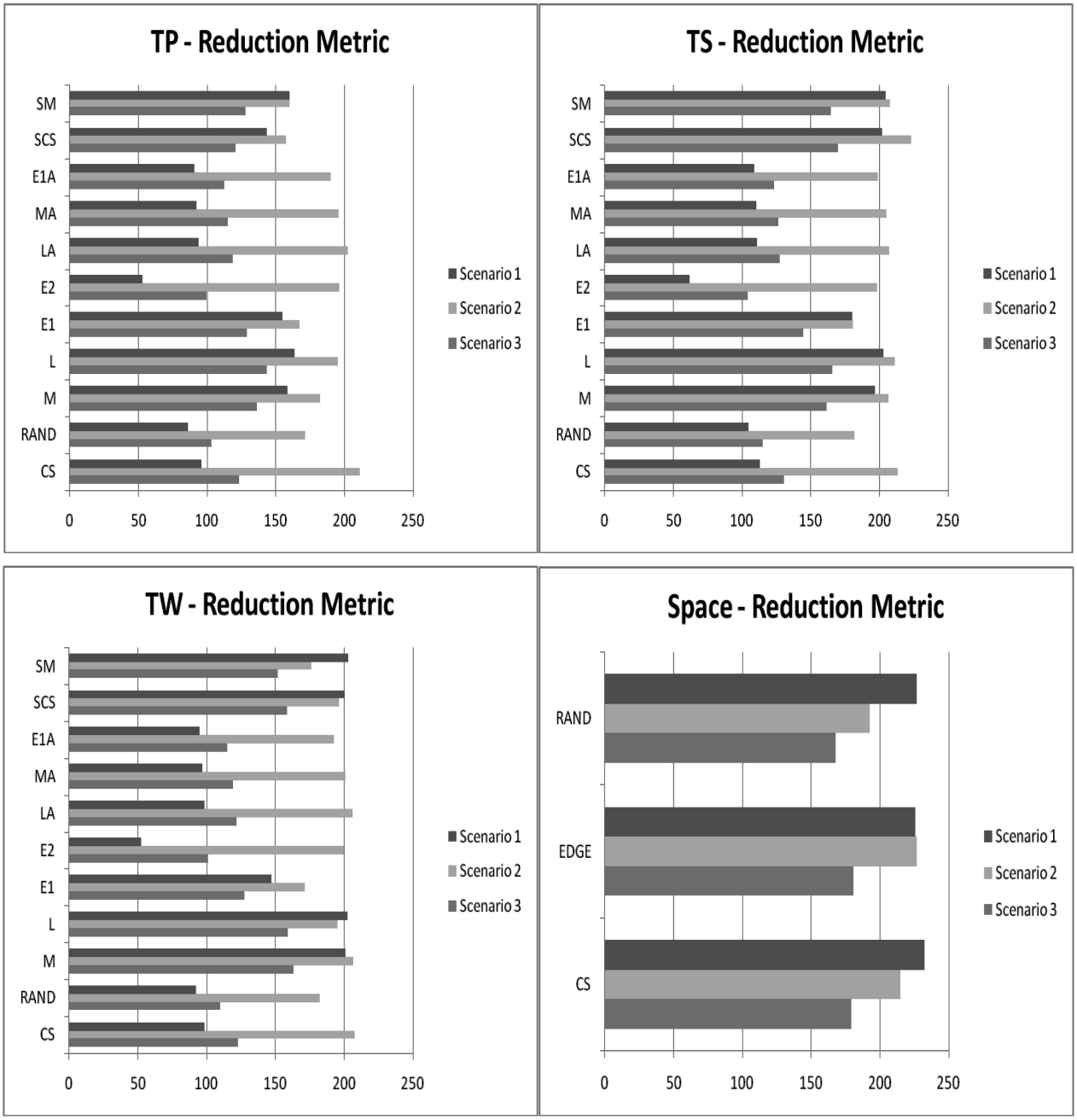

Fig. 8. Average test suite reduction metric over all suite sizes.

techniques MA, LA, and E1A. For *space*, it is interesting to note that, based on the metric, there is very little difference between edges and call stacks when used as reduction criteria in all three weighting scenarios.

Absolute metric values for all three scenarios indicate that test suite reduction, in general, is more effective when applied to TS and TW than in TP. Future work may use this metric in an attempt to identify application construction factors influencing test suite reduction.

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. In addition, 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 [19]. In our 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 that there is a need to better quantify the trade-offs between fault detection effectiveness reduction and size reduction. A cost-benefit model for defect detection activities has been proposed by Wagner [30] and another model specifically focused on regression testing has been developed by Do and Rothermel [3]. Because of the close relationship between regression testing and test suite reduction, Do and Rothermel's model (which explicitly factors in the cost of missing faults and the cost of test execution) may be a good candidate to be applied to the test suite reduction problem. In our future work, we will develop, apply, and evaluate new metrics and cost models to assist practitioners when considering test suite reduction approaches.

ACKNOWLEDGMENTS

Gregg Rothermel provided the space program and test artifacts. Portions of the space package were previously developed by Alberto Pasquini, Phyllis Frankl, and Filip Vokolos. The authors would like to thank Xun Yuan for providing the TerpOffice applications and fault matrices. This work was partially supported by the US National Science Foundation under Grant CCF-0447864 and by the US Office of Naval Research under Grant N00014-05-1-0421.

REFERENCES

[1] G. Ammons, T. Ball, and J.R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* 1997.

[2] W. Dickinson, D. Leon, and A. Podgurski, "Finding Failures by Cluster Analysis of Execution Profiles," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 339-348, 2001.

[3] H. Do and G. Rothermel, "An Empirical Study of Regression Testing Techniques Incorporating Context and Life Cycle Factors and Improved Cost-Benefit Models," *Proc. 14th ACM SIGSOFT Symp. Foundations of Software Eng.,* Nov. 2006.

[4] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Software Eng.,* vol. 28, no. 2, pp. 159-182, Feb. 2002.

[5] P.G. Frankl and O. Iakounenko, "Further Empirical Studies of Test Effectiveness," *Proc. Sixth ACM SIGSOFT Symp. Foundations of Software Eng.,* Nov. 1998.

[6] M. Harder, J. Mellen, and M.D. Ernst, "Improving Test Suites via Operational Abstraction," *Proc. 25th Int'l Conf. Software Eng.,* pp. 60-71, 2003.

[7] M.J. Harrold, R. Gupta, and M.L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Trans. Software Eng. and Methodology,* vol. 2, no. 3, July 1993.

[8] J.R. Horgan and S. London, "Data Flow Coverage and the C Language," *Proc. Fourth ACM Symp. Testing, Analysis, and Verification,* 1991.

[9] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," *Proc. Third Usenix Windows NT Symp.,* pp. 135-143, July 1999.

[10] JavaCCTAgent information on the Web, http://sourceforge.net/projects/javacctagent/, Apr. 2007.

[11] Java Native Interface specification, http://java.sun.com/j2se/1.4.2/docs/guide/jni/, Sept. 2006.

[12] jcoverage information on the Web, http://www.jcoverage.com/, Apr. 2006.

[13] D. Jeffrey and N. Gupta, "Test Suite Reduction with Selective Redundancy," *Proc. 21st IEEE Int'l Conf. Software Maintenance,* pp. 549-558, 2005.

[14] D. Leon and A. Podgurski, "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases," *Proc. 14th IEEE Int'l Symp. Software Reliability Eng.,* Nov. 2003.

[15] S. McMaster and A. Memon, "Call Stack Coverage for GUI Test-Suite Reduction," *Proc. 17th IEEE Int'l Symp. Software Reliability Eng.,* Nov. 2006.

[16] S. McMaster and A. Memon, "Call Stack Coverage for Test Suite Reduction," *Proc. 21st IEEE Int'l Conf. Software Maintenance,* pp. 539-548, 2005.

[17] A. Memon, A. Nagarajan, and Q. Xie, "Automating Regression Testing for Evolving GUI Software," *J. Software Maintenance and Evolution: Research and Practice,* vol. 17, no. 1, pp. 27-64, 2005.

[18] A. Memon, M. Pollack, and M.L. Soffa, "Automated Test Oracles for GUIs," *Proc. Eighth ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 30-39, 2000.

[19] A. Memon, M. Pollack, and M.L. Soffa, "Hierarchical GUI Test Case Generation Using Automated Planning," *IEEE Trans. Software Eng.,* vol. 27, no. 2, pp. 144-155, Feb. 2001.

[20] A. Memon, M.L. Soffa, and M. Pollack, "Coverage Criteria for GUI Testing," *Proc. Eight European Software Eng. Conf./Ninth ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 256-267, 2001.

[21] A. Memon and Q. Xie, "Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software," *IEEE Trans. Software Eng.,* vol. 31, no. 10, pp. 884-896, Oct. 2005.

[22] J. Offutt, J. Pan, and J. Voas, "Procedures for Reducing the Size of Coverage-Based Test Sets," *Proc. 12th Int'l Conf. Testing Computer Software,* pp. 111-123, June 1995.

[23] S. Rapps and E.J. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Eng.,* vol. 11, no. 4, pp. 367-375, Apr. 1985.

[24] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites," *Proc. 14th Int'l Conf. Software Maintenance,* pp. 34-43, Nov. 1998.

[25] G. Rothermel, M.J. Harrold, J. von Ronne, and C. Hong, "Empirical Studies of Test-Suite Reduction," *J. Software Testing, Verification, and Reliability,* vol. 12, no. 4, Dec. 2002.

[26] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, "Test Case Prioritization," *IEEE Trans. Software Eng.,* vol. 27, no. 10, pp. 929-948, Oct. 2001.

[27] A. Rountev, S. Kagan, and M. Gibas, "Static and Dynamic Analysis of Call Chains in Java," *Proc. ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* pp. 1-11, July 2004.

[28] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock, "A Scalable Approach to User-Session Based Testing of Web Applications through Concept Analysis," *Proc. 19th IEEE Int'l Conf. Automated Software Eng.,* pp. 132-141, 2004.

[29] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock, "Web Application Testing with Customized Test Requirements: An Experimental Comparison Study," *Proc. 17th IEEE Int'l Symp. Software Reliability Eng.),* Nov. 2006.

[30] S. Wagner, "A Model and Sensitivity Analysis of the Quality Economics of Defect-Detection Techniques," *Proc. ACM Int'l Symp. Software Testing and Analysis,* July 2006.

[31] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," *Proc. 17th Int'l Conf. Software Eng.,* pp. 41-50, 1995.

[32] T. Xie, D. Marinov, and D. Notkin, "Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests," *Proc. 19th IEEE Int'l Conf. Automated Software Eng.,* pp. 196-205, Sept. 2004.

Scott McMaster received the BS degree in mathematics from the University of Nebraska, Lincoln, in 1994 and the master's degree in software engineering from Seattle University in 2001. He is currently working toward the PhD degree at the University of Maryland, College Park. His research interests include software testing, program analysis, software tools, and distributed systems. He is a member of the ACM and the IEEE Computer Society.

Atif M. Memon received the BS degree from the University of Karachi in 1991, the MS degree from the King Fahd University of Petroleum and Minerals in 1995, and the PhD degree in computer science from the University of Pittsburgh in 2001. He received fellowships from the Andrew Mellon Foundation for his PhD research. He is currently an associate professor in the Department of Computer Science at the University of Maryland, College Park. His research interests include program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He is a member of the ACM and the IEEE. He received a gold medal during his undergraduate studies and the US National Science Foundation Faculty Early Career Development (CAREER) Award in 2005.