ABSTRACT

Title of Document:	A CONTEXT-SENSITIVE COVERAGE CRITERION FOR TEST SUITE REDUCTION
	Scott David McMaster, Doctor of Philosophy, 2008
Directed By:	Professor Atif Memon, Department of Computer Science, University of Maryland, College Park

Modern software is increasingly developed using multi-language implementations, large supporting libraries and frameworks, callbacks, virtual function calls, reflection, multithreading, and object- and aspect-oriented programming. The predominant example of such software is the graphical user interface (GUI), which is used as a front-end to most of today's software applications. The characteristics of GUIs and other modern software present new challenges to software testing. Because recently developed techniques for automated test case generation can generate more tests than are practical to regularly execute, one important challenge is *test suite reduction*. Test suite reduction seeks to decrease the size of a test suite without overly compromising its original fault detection ability. This research advances the state-of-the-art in test suite reduction by empirically

studying a coverage criterion which considers the *context* in which program concepts are covered. Conventional approaches to test suite reduction were developed and evaluated on batch-style applications and, due to the aforementioned considerations, are not always easily applicable to modern software. Furthermore, many existing techniques fail to consider the context in which code executes inside an *event-driven* paradigm, where programs wait for and interactively respond to user- and system-generated *events*. Consequently, they yield reduced test suites with severely impaired fault detection ability. The novel feature of this research is a test suite reduction technique based on the *call stack* coverage criterion which addresses many of the challenges associated with coverage-based test suite reduction in modern applications. Results show that reducing test suites while maintaining call stack coverage yields good tradeoffs between size reduction and fault detection effectiveness compared to traditional techniques. The output of this research includes models, metrics, algorithms, and techniques based upon this approach.

A CONTEXT-SENSITIVE COVERAGE CRITERION FOR TEST SUITE REDUCTION

By

Scott David McMaster

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor Atif Memon, Chair
Professor Adam Porter
Professor James Purtilo
Professor Ramani Duraiswami
Professor Gang Qu

Acknowledgements

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 Galileo group at the University of Nebraska - Lincoln provided the nanoxml program and test artifacts, and Alex Kinnear provided valuable assistance in working with nanoxml. I would like to thank Xun Yuan for providing the TerpOffice applications and fault matrices, and Jaymie Strecker for her work on the GUITAR benchmark web site.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1. Automated Test Case Generation Landscape

Interest in the development and application of automated test case generation techniques has grown in recent years. The growing complexity of modern software applications has piqued test engineers' interest in leveraging these new approaches to improve software quality. And the reduced cost and increased availability of high-performance hardware has expanded the range of techniques that can be implemented in practice. Easy-to-use commercial tools such as those by Parasoft [32] and Agitar [1] can automatically generate unit tests based on C++ and Java source code, and model-based techniques can generate tests from UML diagrams [22] or maps of graphical user interfaces [29]. Most automated test case generation approaches share one common characteristic when applied to non-trivial software applications: Specifically, they generate a large quantity of tests.

At the same time, the software development and release lifecycle is growing shorter. Market demands are pushing practitioners toward "agile" development processes that include nightly builds and continuous integration [3]. These processes usually mandate regular automated testing. However, if test suites are too large, the time it takes to run them can be the longest, most inefficient step of the process. This can discourage engineers from taking full advantage of the aforementioned automated test case generation techniques.

1.2. Test Suite Reduction

For this reason, the problem of *test suite reduction* [13][42][36] is interesting and relevant. Test suite reduction 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 [36], dataflow [42], or dynamic program invariants [12]. In such an approach, each *coverage requirement* (*i.e.*, 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, batch-driven software applications such as parsers and interpreters. Test cases for these applications are generally built by partitioning the input space into equivalence classes and selecting one or more inputs from each class, along with test cases to cover boundary conditions.

Of particular interest is how test suite reduction techniques perform when applied to modern software applications. Consider the current leading paradigm for user interaction, the *graphical user interface* (GUI). 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 dramatically influence 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 [28]. Modern approaches often leverage sophisticated models of the application under test to generate test inputs. For

example, a recent test-case generation technique based on *event-flow coverage* has been shown to be effective for defect detection in GUI applications [29]. 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.

1.3. Call-Stack-Based Test Suite Reduction

This research develops a novel approach to test suite reduction based on the call-stack coverage criterion. 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 the simple Java program in Figure 1(a) appears in Figure 1(b).

```
public class HelloWorldApp {

   public static void main(String[] args) {

              System.out.println("Hello World!");

           }

        }
```

```
(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: (a) A Hello-world Example and (b) Associated Call Stack

This call stack was collected by tools developed in support of this research. In Figure 1(b), 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 println method call seen in Figure 1(a). 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,

specifically, the sequence of active method calls. Besides 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 additional advantages for modern software applications in the areas of libraries and frameworks, object-oriented language features, multithreading, and multi-language implementation. These advantages are discussed in detail in the following sections.

This research shows that the call stack coverage criterion provides effective tradeoffs between size and fault detection effectiveness for modern software applications when applied to the problem of test suite reduction.

1.4. Test Suite Reduction Challenges Addressed by Call Stacks

Modern software poses new challenges for coverage-based testing that require the development of new solutions. For example, the execution model for a GUI, based on an event-listener loop, differs from that of conventional or batch-driven software. During GUI execution, users perform actions which 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 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 software applications increasingly integrate multiple source code languages and object code formats. They are developed using new programming languages utilizing object-oriented or aspect-

oriented paradigms. They make use of virtual function calls, reflection, multithreading, and event handler callbacks. Taken together, these features severely impair the applicability of techniques that rely on static analysis or the availability of language- and/or format-specific instrumentation tools.

1.3.1. 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 requirement 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 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 2. If no library code is instrumented, every execution of this program against integral input will satisfy line, branch, and dataflow coverage. Thus, when used in test suite reduction, each of those coverage approaches could potentially drop all tests that exercise the code with integral input greater than or less than zero, thereby missing coverage of the array-index-out-of-bounds exception that occurs with such input. In contrast, the call stack coverage technique presented in this research 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.

```
public class ArrayTest {

   public static void main(String args[]) {

      String[] strings = {"first"};

      int index = Integer.parseInt( args[0] );

      System.out.println( strings[ index ] );

   }

}
```

Figure 2: A Simple Program Demonstrating the Impact of Library Code on Errors

1.3.2. Object-Oriented Language Features

Modern GUI application frameworks, usually implemented in languages like C++, Java, or 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 Figure 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 pass 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 behavior that should be captured and preserved after test suite reduction. But static analysis cannot in general determine that *toUpperCase* or *toLowerCase* may be invoked by this program. Similarly, 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.

```
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() );
        }
}
```

Figure 3: A Simple Example Demonstrating the Impact of OOP Features on Errors

1.3.3. Multithreading

Most modern software runs with multiple threads of execution. Indeed, current GUI applications are all multithreaded: Minimally, there is one thread listening for user actions and another thread executing events. And all Java and .NET

applications are multithreaded, if for no other reason than the presence of the garbage collector. Multiple threads of execution present challenges for traditional coverage techniques, which have typically been conceived for a sequential model [44]. For example, when collecting def-use coverage at runtime, it is not clear how to associate the use of a variable with a single definition when definitions can occur on multiple threads.

Call stack coverage is fundamentally a sequential criterion. However, as will be discussed in Chapter 3, it is straightforward to define an approach to collecting call stack coverage that is both simple and efficient to execute in a multithreaded environment.

1.3.4. Multi-Language Implementations

Many traditional coverage criteria depend on the ability to fully instrument the source or object code of an application. In a multi-language implementation, the necessary tools to insert this instrumentation may not exist for all source languages or object code formats in use. Moreover, any tools that *are* available across technologies may not be interoperable in such a way to enable collection of complete coverage data. Unlike coverage based on these criteria, call stack coverage is easily captured in a multi-language application, and with or without the availability of source code. In general, writing a tool to collect call stacks only requires method entry and exit hooks, which already exist inside most compilers or runtime platforms to enable the construction of call profilers. A large application implemented in multiple languages is no different from a single-language implementation when abstracted via the run-time call stack.

1.5. Test Suite Reduction Metrics

Test suite reduction techniques are traditionally evaluated based on how small the reduced suites are and how effective they are at detecting a set of known faults [[42][36]]. Because the ideal reduced test suite – a single test case that detects all faults – is not generally obtainable, practitioners are left to evaluate the research data and pick the most appropriate tradeoff between the size reduction and fault detection reduction metrics. To make a more informed decision, practitioners would benefit from different ways of looking at this tradeoff. To assist in this matter, this research also develops a new weighted single-point metric for test suite reduction and applies it to the empirical results. Additionally, no existing test suite reduction metric explicitly factors in test coverage data to account for and attempt to explain the performance of a given technique. To remedy this situation, this research defines a new metric based on the *average expected probability of finding each fault* in a reduced test suite.

1.6. Implementation and Evaluation

To enable empirical studies of call-stack-based test suite reduction, a number of tools and analyses have been implemented. These will be discussed in detail in Chapter 4. Briefly, the tools include libraries for capturing call stacks from a running software application on two different platforms, along with an implementation of an existing test suite reduction heuristic. Several programs were implemented to analyze coverage data and calculate metrics. These metrics include the traditional test suite reduction metrics of percentage size reduction and percentage fault detection reduction, as well as the additional metrics proposed and developed by this research.

Additionally, this research has resulted in four publications to date. Initial work with a conventional subject application on using call stacks as a coverage criterion in test suite reduction was presented at the International Conference on Software Maintenance (ICSM) in 2005 [24]. This work showed that the call stack coverage criterion provided good tradeoffs between test suite size reduction and loss of fault detection effectiveness. The call stack approach was then targeted at modern subject applications in work presented at the International Symposium on Software Reliability Engineering (ISSRE) in 2006 [23]. This work showed that call-stack-based test suite reduction is particularly effective in modern GUI-based software applications. Expanded work from those conference papers has been accepted for journal publication in IEEE Transactions on Software Engineering (TSE). The TSE paper includes comparisons of call-stack-based reduction to additional types of reduced test suites, and it incorporates new analyses. A novel analysis approach developed by and key to this research was presented at the International Conference on Software Maintenance (ICSM) in 2007 [25]. This approach, based on the average probability of detecting each fault in a reduced test suite, has applicability to the general problem of test suite reduction.

1.7. Contributions

This research makes the following contributions to the fields of test suite reduction and software testing:

1. It defines and develops *call stacks* as a coverage criterion for use in test suite reduction.

2. It empirically evaluates call stacks in the context of coverage-based test suite reduction versus several traditional coverage criteria.
3. It investigates the importance of including library and framework coverage information when reducing test suites.
4. It empirically shows that the effectiveness of test suite reduction techniques can differ between conventional and modern software applications.
5. It develops a new weighted single-point metric for effectiveness of test suite reduction techniques to be applied by practitioners considering test suite reduction.
6. It analyzes coverage-based test suite reduction techniques using a novel metric that explicitly accounts for test coverage data and the *average expected probability of detecting each fault* in a reduced test suite.
7. As an effect, it produces tools and analyses that can be used by other researchers in furthering the study of call stacks and test suite reduction.
8. It produces data including program artifacts, full and reduced test suites, fault matrices, and coverage data which can be made available to other researchers to aid in their investigations of test suite reduction in particular and test case management problems in general.

The rest of this document is structured as follows. Chapter 2 discusses related work. In Chapter 3, a formal model for call stacks is defined. Chapter 4 presents the tools and techniques developed and used in this research. In Chapter 5, existing and novel metrics for the evaluation of coverage-based test suite reduction are discussed. Chapter 6 presents a series of experiments to answer research questions related to the

use of call stacks in test suite reduction. In Chapter 7, the results are analyzed relative to a newly proposed metric for test suite reduction. And Chapter 8 concludes and discusses future work in this line of research.

Chapter 2: Related Work

Several approaches to the problem of test suite reduction have been proposed by other researchers. Many of those employ test coverage information to determine which test cases should remain in a reduced suite and which should be discarded. Key problems remain with traditional approaches, including the challenge of collecting various types of coverage data in modern software applications and limitations in the tradeoff between size reduction and fault detection effectiveness – the two metrics against which test suite reduction has traditionally been evaluated.

This work is particularly concerned with developing new coverage criteria for modern software applications. Many applications that employ *graphical user interfaces* (GUIs) exemplify the characteristics of modern software that motivate this research, including object-orientation, extensive use of libraries, and multithreaded execution. Other researchers have developed approaches to the general problem of GUI testing which will be used in this work.

This research applies call stack coverage to the problem of test suite reduction. Several researchers have developed other types of program analyses that leverage sequences of method calls.

Related research from the areas of GUI testing, test suite reduction, and call chains are presented here.

2.1. Test Suite Reduction

There have been numerous studies of test suite reduction while holding coverage constant relative to some criterion and evaluating reduction's relationship to

fault detection effectiveness. Wong *et al.* [42] 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.* [35] reduce with respect to all-edges coverage and find significant reductions in fault detection effectiveness. They contrast their results with those of Wong *et al.* [42] and suggest possible causes for the different conclusions. However, collecting all-uses and other dataflow 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 [15]. In contrast, call stack coverage information is relatively simple to obtain using tools developed and made available as a part of this research [17]. Additionally, call stack coverage can be analyzed on any stack-based runtime environment, which encompasses most language and system combinations in practical use today.

To develop and evaluate the idea of call-stack-based test suite reduction, this research uses the *ReduceTestSuite* heuristic presented by Harrold *et al.* [13]. This heuristic is discussed in more detail in Section 4.2. A different approach to coverage-based test suite reduction known as the "ping-pong" heuristics is given by Offutt *et al*. [31]. Using the "ping-pong" heuristics in call-stack-based reduction is a possible avenue of future work.

There are alternative approaches to test suite reduction that do not explicitly maximize test coverage relative to a traditional criterion. One such alternative is the "operational difference" technique of Harder *et al*. [12]. This approach builds up a

reduced suite by pulling test cases from the test pool and adds them to the suite if they change the "operational abstraction," which is a mathematical picture of the program's dynamic behavior maintained across the execution of the test set. This process terminates when a certain number of consecutive cases produce no abstraction changes. Another approach that does not explicitly attempt to maximize test coverage is the cluster sampling of Leon and Podgurski [21], and Dickinson *et al.* [6]. The average probability of detecting each fault as defined in this research could be used to identify the best coverage criteria to be used as inputs for cluster formation. Also, the context-preserving nature of call stack coverage should make it an excellent criterion on which to cluster test cases.

Jeffery and Gupta [20] introduce a test suite reduction approach that combines "primary" and "secondary" coverage criteria in the reduction algorithm. The "selective redundancy" technique is so named because certain test cases are known to add no additional coverage of the primary criterion, but by selecting such tests based on the second criterion, they are able to generate reduced test suites with fault detection effectiveness better than using either criterion alone. Results from this research for the average probability of detecting each fault when using pairs of coverage criteria provide some additional evidence that combining criteria can be particularly effective in test suite reduction. Additionally, call stack coverage would be an interesting choice as a criterion in this technique, perhaps as a secondary participant with one of the simpler but context-insensitive criteria such as statement or branch coverage.

Sampath *et al.* use concept analysis to generate minimal test suites from user sessions defined as URLs in a web application [39]. 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.

In their study of test suite reduction for model-based tests, Heimdahl and George raise the notion of an "ideal coverage criterion" which "would detect *all* faults in the system under test and *any* test-suite, large or small, providing this coverage would reveal the same faults" [14]. Along the same line, Rothermel *et al.* point out that assuming an equal likelihood of selecting one of k test cases that hit a coverage requirement, and only one test case detects a given fault, the probability of omitting the fault-detecting test case under coverage-based test suite reduction is $(k-1)/k$ [36]. This research claims to be the first to attempt to formalize and fully quantify these notions.

The test suite reduction problem is closely related to *test case prioritization [10]*, because any reduction technique can be turned into a prioritization technique by repeated application of the reduction algorithm to the remainder of the suite.

2.2. GUI Testing

This research conducts empirical studies to evaluate the effectiveness of the call stack coverage criterion in test suite reduction compared to other possible approaches. When considering other approaches, it is notable that new coverage

techniques for GUIs have recently been developed. Event-based coverage [29] 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 this research use two different event coverage criteria, "event" and "event-interaction" [29]. In event coverage, each event in isolation is a coverage requirement, while in event-interaction coverage, unique pairs of events are included as requirements.

Empirical testing studies of GUI applications are aided by the availability of the GUITAR infrastructure [30]. GUITAR includes several subject applications along with fault-seeded versions, a universe of test cases, and fault matrices mapping test cases to the faults that they uncover. This infrastructure also includes a test case automation runtime, the JavaGUIReplayer, which makes it possible to rapidly and automatically execute test suites against the subject applications for the purpose of collecting test coverage data.

2.3. Call Chains

Rountev *et al.* [38] also consider the problem of "call chain" 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 this research, the authors do not address the impact of this type of coverage on test suite reduction, and their dynamic analysis assumes exception-free, single-threaded execution.

The Rostra framework [43] 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, the call stack approach used in this research operates on an entire program rather than individual objects. Rostra is therefore focused on and only appropriate for unit-level testing. This research shows that call stacks can be effective at the system level. Also unlike Rostra, the call stack technique used in this research makes no assumptions about the threading behavior of test case executions or the usage of shared variables.

This research makes use of the calling context tree (CCT) data structure to collect call stack coverage data. The calling context tree provides an efficient approach to track the context of method calls in a running program. Ammons *et al.* first proposed the calling context tree and provided a deterministic algorithm for building it at runtime [1]. Bond and McKinley present a probabilistic method of approximating the calling context tree which can be more efficient [4].

2.4. Summary

Prior to this research, call stacks had not been used before as a criterion for coverage-based test suite reduction. This approach advances the state of the art in test suite reduction in three primary ways. First, call stack coverage data is simple and efficient to collect in most runtime environments, and yet it still captures much interesting dynamic program behavior, including the context in which method calls occur. Second, empirical results of this research show that call-stack-based test suite reduction provides unique and potentially desirable tradeoffs between reduction in size and reduction in fault detection effectiveness. Finally, evaluating the

effectiveness of call-stack-based test suite reduction while performing this research has led to the development of new metrics and analyses which can be applied by other researchers in future studies.

Chapter 3: Modeling and Collecting Call Stacks

This research is grounded in a robust formal model of call stacks which considers how call stack coverage data can be collected, and how that data is applied to the problem of test suite reduction. The following sections develop this model and its motivations.

3.1. Considerations in Modeling Call Stacks

3.1.1. Runtime Feasibility

There are multiple ways to collect call stack coverage data. The most naïve approach is to collect from a running program a full trace of method calls and returns. Later, the set of call stacks can be trivially constructed offline by stepping through this trace. This approach also has the benefit that no sophisticated in-memory data structures must be maintained during program execution. In practice, this method proves infeasible for non-trivial programs and test cases as the size of the traces grows linearly with the length of the test cases, causing the resulting data sets to become awkward to store and post-process.

A second approach is to capture a snapshot of the current call stack at each method call in the running program and add it to a set of all stacks observed during the test case. Compared to method traces, the volume of coverage data produced by this technique will scale better with the length of the test case as, over time, many stacks will be observed repeatedly. However, a disadvantage of this approach is that maintaining the set of *all* stacks observed is relatively memory- and CPU-intensive at

runtime, as the stacks must be stored and available online, and newly observed stacks must be checked against those already observed.

The problems with the second approach are largely due to the fact that it maintains a large amount of redundant data. The observation that pushing a new method onto the runtime call stack (to make it "taller") is actually just an extension of the ("shorter") call stack that was current immediately prior leads to a more efficient approach. In the third approach, only the unique *deepest* stacks are maintained, giving the same knowledge about the call stack coverage generated by a test case but with far less data volume and runtime overhead. Sections 3.2 and 3.3 describe this approach in detail.

All of the above approaches to collecting call stack data can easily be extended to apply to multithreaded programs by collecting coverage per thread of execution and merging the data after threads exit. However, call stacks are intrinsically tied to a single thread and thus do not explicitly capture any information about thread interactions.

3.1.2. Representation of Call Stacks

There are also multiple ways to model and represent call stacks for use in test suite reduction. In Figure 1(b), 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 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 and effectiveness of the reduction technique. For coverage-based test suite reduction to work well, neither too many nor too few unique coverage requirements should be observed by a full test suite. Some models may generate so many distinct call stacks that too little redundancy exists to serve as a basis for eliminating test cases. Other models may generate so few different call stacks that differences between test cases are lost, leading too many test cases deemed redundant and therefore discarded. In such a scenario, fault detection effectiveness is compromised.

Additionally, collection and analysis for a highly granular model (such as one including method parameter values) may be infeasible from a resource perspective. 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, when using full method signatures, a typical execution of the simple program in Figure 1(a) generates 803 call stacks; subject applications built with Java Swing and used in the experiments of Chapter 6 generate hundreds of thousands. One possible approximation to complete call stack coverage which is far less resource-intensive is to omit library calls from the collected call stacks. Techniques with and without library call information are considered in the experiments of Chapter 6.

3.2. 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 $c = \langle m_1, m_2, \ldots m_n \rangle$

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:

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

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

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 $C(t)^s$. This research defines the set of deepest, or *maximum depth*, stacks $C(t)_{max}$ in a thread *t* as follows:

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

Here, $\varnothing$ 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, it is possible to combine call stack observations made on each thread that took part in a given program execution. Thus, the set of threads that existed during execution is defined as:

$$(4)\ \ T = \langle t_1, t_2, \ldots t_n \rangle$$

The set of unique call stacks for a program input *I* is represented by:

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

$C_{max}(I)$ 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 the 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 the technique to produce less size reduction than it might otherwise, it is allowed 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.

This research defines 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*. Two test cases tc_1 and tc_2 are considered to be equivalent if they generate identical sets of maximum depth call stacks.

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

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

$$(7) \qquad Stacks(TS) = \cup\{C_{max}(tc) \mid tc \in TS\}$$

A *test suite reduction technique* is defined to be a complete approach for reducing the size of a test suite, including any necessary static or dynamic program analysis. For coverage-based test suite reduction, a technique consists of a coverage criterion and an algorithm for reducing the suite while holding coverage of that criterion constant. The proposed technique considers a maximum depth call stack to be a *coverage requirement* in the test suite reduction algorithm *ReduceTestSuite* [13].

Thus, execution of a reduced test suite $TS^{reduced}$ will generate the same set of unique call stacks as execution of its original (full) counterpart TS^{full}, *i.e.*, $Stacks(TS^{full}) = Stacks(TS^{reduced})$.

3.3. Calling Context Tree

An efficient data structure for recording call stacks on a given thread of execution is the *calling context tree*, or 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 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:
    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 non-trivial applications, the size of the CCT data structure does not grow unbounded (as a full method trace would) over the run-time

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.

A CCT-based approach to collecting call stack coverage is easily extensible into a multithreaded environment. One approach would be to maintain a single CCT shared and updated by all threads in a multithreaded program. Synchronization of access to this data structure becomes an issue, however. An alternative approach (and the approach used in this research) 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 single-CCT alternative. 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 the applications or test cases used in this research.

3.4. Summary

By showing how the problem of collecting call stack coverage data is equivalent to computing the set of unique maximum-depth stacks for a given program input, and applying the CCT data structure at runtime, it becomes possible to efficiently determine call stack coverage of a test suite. The following chapter presents a concrete implementation of these ideas which is suitable for experimentation.

Chapter 4: Implementation

No previously existing tools were found suitable to collect call stack coverage data from a running program in any environment. Therefore, to conduct research into call-stack-based test suite reduction, it was necessary to build several tools from scratch. Most important is JavaCCTAgent, a tool to collect call stacks from a running Java program that has been made available to the research community [17]. In the following sections, all of the tools will be discussed in detail.

4.1 Collecting Call Stacks

4.1.1. General Approach

One of the key advantages to using call stack coverage as opposed to other types of coverage is that very little instrumentation or platform support is required to collect call stacks from a running program. All that is strictly necessary is the ability to be dynamically notified when a method is called and when it returns, so that the proper state of a calling context tree (CCT) can be maintained. These same operations are fundamental to the operation of call profilers, and therefore they have long been readily available in most language, environment, and platform combinations. Indeed, both call stack coverage collection libraries used in this research operate with no modification to the original program source.

As discussed in Section 3.3, collection of call stack coverage requires that a CCT must be maintained for each thread. Therefore, it is also important for a call stack coverage implementation to be notified by the environment when threads are

created and destroyed. In the multithreaded environment used in this research (Java), this requirement was well-supported.

4.1.2. Detours-Based Implementation for Win32

Two concrete implementations of the general approach were used in this research. The first works on C/C++-language programs on the Windows platform, making use of the Detours package [16]. Detours is a library that allows dynamic interception of binary function calls on the Win32 platform without modifying the on-disk program. Detours' "dynamic trampoline" functionality is used to insert hooks at each function entry and exit in the application-under-test to build the CCT. This approach requires specific instrumentation code external to the target program for each function in the program, and the use of a binary version containing debugging symbols. This instrumentation code was generated by a tool whose input was a list of function prototypes. The generated code was built into a separate code module attached to the subject application's process at runtime using functionality in Detours. Thus, neither the source code nor the on-disk program of the subject application is modified. This implementation was not used on any recursive programs and therefore has no support for recursion built into the CCT module.

Since it is expected that Win32 programs will make use of the Standard C Library, the instrumentation of that code is also addressed. Instead of instrumenting all public and internal functions in the library (which would require examination of the full library source code to make use of a Detours-based approach), the implementation used in this research only tracks those functions defined in the public C library headers and called by an application under study or a macro used by an

application under study. Thus, internal library functions do not appear on the call stacks collected by this implementation, making them in fact an approximation (albeit a good one since most C library function implementations do not generate deep call graphs). As discussed in Chapter 3, there is a tradeoff between the level of detail included in the call stacks (and thus the effectiveness of the technique) on one hand and the practicality of instrumentation and analysis time on the other. The limitations inherent in the Detours-based approach served as a motivation to do an implementation for the Java environment, where the impact of library functions and the fidelity of call stacks could be studied in detail.

4.1.3. JVMTI-Based Implementation for Java

The second implementation for call stack coverage data collection targets Java programs. The Java Virtual Machine environment has advantages over the Detours/Win32 environment for the study of call stacks. By building a Java Virtual Machine Tool Interface (JVMTI) agent [33], it is a simple matter to collect call stack coverage data from the entire stack of runtime libraries. Additionally, experimental artifacts more representative of the modern techniques that motivate this research were available for Java.

JavaCCTAgent was built as a part of this research to collect the CCT data necessary for call stack coverage analysis [17]. In this implementation, call stacks are represented as an ordered set of full method signatures of the active methods. The JVMTI hooks for method entry and method exit are used to maintain a CCT for each thread. Direct recursive invocations are permitted in this tool but are only captured to a depth of one. As threads die and at the end of program 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.

Since coverage is collected for each thread, data on system threads is being collected 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 serves 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 the data collection and, by consequence, may have 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) [18] 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 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.

4.2. Reducing Test Suites

This research uses the *ReduceTestSuite* algorithm presented by Harrold *et al.* [13] to reduce a full test suite given its coverage information. Because finding a minimal test suite that satisfies each coverage requirement is an NP-complete

problem [13], *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 runtime of this algorithm is $O(n*\text{Max}(m,n))$. The implementation of *ReduceTestSuite* used in the subsequent empirical studies is written in C#.

4.3. Other Tools

Additional tools were utilized in the experiments to execute test cases and collect various types of coverage. For GUI applications, a tool called GUI Ripper [26] automatically derives a model of a GUI, and from that model, test cases with varying event sequence lengths can be automatically generated. Another tool, the JavaGUIReplayer [30], can subsequently be used to execute the test cases.

In this research, line coverage data was obtained using the jcoverage tool [19] or the very similar Cobertura tool[1] [5]. For feasibility, the line coverage technique does not include coverage of supporting libraries for Java programs, but rather only includes coverage of the subject application source.

[1] Over the course of this research program, jcoverage ceased to be freely available. Cobertura provides equivalent functionality but is open-source.

Chapter 5: Test Suite Reduction Metrics

A primary concern of this research is determining how call stacks compare to other coverage criteria when used in coverage-driven test suite reduction. The effectiveness of applying various coverage criteria in test suite reduction is traditionally based on empirical comparison of two metrics derived from the full and reduced test suites and information about a set of known faults. The two metrics, which follow directly from the dual goals of test suite reduction, are percentage size reduction and percentage fault detection reduction. Additionally, to further validate the usefulness of call stack coverage, this research seeks a deeper understanding of why a given criterion performs well or poorly in the test suite reduction problem. Along those lines, a novel contribution of this research is a new *fault detection probability* metric.

5.1. Percentage Size Reduction

Percentage size reduction is a direct measure of the number of test cases that are eliminated from a full test suite by a reduction technique. Given the sizes of a full and corresponding reduced test suite, the value is given in Equation (8):

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

5.2. Percentage Fault Detection Reduction

Percentage fault detection reduction measures the percentage of faults found by a full test suite that are not found by the corresponding reduced test suite. In this research, fault detection effectiveness is measured 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 by Rothermel *et al.*[35] and Wong *et al.*[42]. Given the number of faults detected by a full and corresponding reduced test suite, the value is given in Equation (9):

(9) % Fault Detection Reduction = $100 * (1 - \text{FaultsDetected}_{\text{Reduced}} / \text{FaultsDetected}_{\text{Full}})$

Other researchers sometimes use *fault detection effectiveness* as an alternative to percentage fault detection reduction. Fault detection effectiveness measures the percentage of faults *retained* rather than lost in a reduced test suite. Therefore:

(10) Fault Detection Effectiveness

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

= 1 - % Fault Detection Reduction

5.3. Fault Detection Probability Metric

Neither the percentage size reduction nor the percentage fault detection reduction explicitly factors test coverage data into the calculation. This limits the usefulness of these metrics to account for and attempt to explain the performance of a given coverage criterion and technique. This section defines a number of functions on the coverage and fault data collected from an application, its test pool, a set of known faults, and a coverage criterion. These definitions lead to a new metric for coverage-based test suite reduction utilizing the *average probability of detecting each fault*. Intuitively, this metric captures the likelihood that coverage-preserving reduced test suites will detect the same faults as their original counterparts, taking into account the number of coverage requirements which only appear in fault-detecting test cases. Subsequent experiments in Chapter 6 will show that this quantity varies greatly

depending on the selected coverage criterion, thus making it useful in selecting the best criterion to use in a test suite reduction technique.

5.3.1. Data Structures

Given a subject application, a set of test cases *TC(1..J)*, a set of known faults *KF(1..K)*, and a set of coverage requirements *CR(1..I)*, it is possible to obtain two artifacts important to the study of test suite reduction, as well as the closely related topics of test case prioritization [37] and regression test selection. The first is the *coverage matrix, C*, [9] for a test suite. In a coverage matrix, each row represents a coverage requirement, such as a line, edge or call stack, and each column represents a test case. A cell value *C(i, j)* is 1 if coverage requirement *i* is satisfied by test case *j* and 0 otherwise. Based on the coverage matrix, it is possible to define a function *covReqTCs(C, i)*, which, given a coverage matrix *C* and a coverage requirement *i*, returns the set of test cases which satisfy the given requirement.

$$(11) \quad \text{covReqTCs}(C,i) = \{j \in TC \mid C(i, j) = 1\}$$

Second, consider the fault matrix, F, where each row represents a known fault and each column is a test case. A cell value *F(k, j)* is 1 if fault *k* is detected by test case *j* and 0 otherwise. This leads to another function, *detectsFaultTCs(F, k)*, which accepts a fault matrix *F* and fault number *k* and returns the set of test cases that detect *k*.

$$(12) \quad \text{faultDetectingTCs}(F,k) = \{j \in TC \mid F(k, j) = 1\}$$

For a given test suite, the matrices *C* and *F* have the same column rank which is the number of test cases.

5.3.2. Metric Definition

Making use of the coverage matrix *C* and fault matrix *F*, this research defines a metric that captures the average expected probability of finding each fault after coverage-based test suite reduction. This metric will be independent of the selection of a specific coverage-preserving reduction algorithm. From *C* and *F*, the *fault correlation* for a coverage requirement *i* to a fault *k* is defined as the ratio of test cases in the test suite that satisfy the coverage requirement *and* detect the fault to the number of test cases that merely satisfy the coverage requirement. This value is calculated from the cardinality of these sets as follows:

$$\text{faultCorr}(C,F,i,k) = \text{Card}[\ \text{covReqTCs}(C,i) \cap \text{faultDetectingTCs}(F,k)\ ] \ / \ \text{Card}[\ \text{covReqTCs}(C,i)\ ] \quad (13)$$

If a coverage requirement *i* is satisfied only by test cases that detect a given fault *k*, then $\text{faultCorr}(C,F,i,k) = 1$, the maximum possible fault correlation. Intuitively, any coverage-preserving test suite reduction technique *must* select a fault-detecting test case for that fault.

If a coverage requirement is satisfied by two test cases, one of which detects a given fault and one of which does not, the fault correlation with that coverage requirement is 0.5. If no coverage requirement leads to a higher fault correlation, then a coverage-preserving test suite reduction technique would select a fault-detecting test case with a minimum probability of 0.5. The cumulative effect of fault correlations from other coverage requirements may further raise the actual probability of detecting the fault if those requirements are covered by different test cases than the coverage requirement with maximum fault correlation. But for simplicity, consider

each coverage requirement and corresponding fault correlation independently, which reflects the worst-case scenario. Then, the expected (minimum) probability of finding a given fault after test suite reduction is defined as the maximum fault correlation of all coverage requirements with that fault:

(14) $\text{expProbFindFault}(C,F,k) = \text{Max}(\ \text{faultCorr}(C,F,i,k),\ i \in CR)$

This definition can be extended to incorporate all known faults as follows: The expected probability of finding all faults after test suite reduction is the product of the expected probability of detecting each fault:

(15) $\text{expProbFindAll}(C,F) = \Pi(\ \text{expProbFindFault}(C,F,k),\ k \in KF)$

Because a goal of this research is to compare how various coverage criteria (call stacks in particular) perform in test suite reduction, a metric which is normalized across subject applications and test suites with differing numbers of coverage requirements and detectable faults is required. Thus, the metric that will be considered is the *average expected probability of detecting each fault*:

(16) $\text{avgExpProbFindEach}(C,F) = \text{Avg}(\ \text{expProbFindFault}(C,F,k),\ k \in KF\ )$

Figure 4 presents an algorithm for calculating (16) for a given subject application, fault matrix, and coverage matrix.

```
ALGORITHM:  CalcFaultDetectionProbability (
1  C(1..I, 1..J), /* coverage matrix, I=number of
       coverage requirements,  J=number of test cases*/
2  F(1..K, 1..J) /* fault matrix, K=number of known
                    faults, J=number of test cases */
3    Declare P(1..K) /* expected probabilities of
                        finding faults 1..K */
4    for k = 1..K { /* for each fault */
5      P(k) = 0
6      for i = 1..I { /* for each coverage
                         requirement */
7        countCoveringCases <- 0
8        countCoveringDetectingCases <- 0
9        for j = 1..J { /* for each test case */
10         if C(i, j) = 1 then {
11           countCoveringCases <-
                          countCoveringCases + 1
12         }
13         if F(k, j) = 1 then {
14           countCoveringDetectingCases <-
                 countCoveringDetectingCases + 1
15         }
16       } /* j */
17       if countCoveringCases = 0 then next i
18       faultCorrelation =
                countCoveringDetectingCases /
                        countCoveringCases
19       P(k) = Max(faultCorrelation, P(k))
20     } /* i */
21   } /* k */
22   Return Sum(P(1..K)) / K
```

Figure 4: CalcFaultDetectionProbability Algorithm

The CalcFaultDetectionProbability algorithm assumes the coverage matrix and fault matrix as inputs (Lines 1 and 2). It then declares an array with length equal to the number of known faults to hold the calculated probabilities (Line 3). Then for each coverage requirement for each fault, counters are initialized to hold the number

of test cases that cover the requirement, and both cover the requirement and detect the fault (Lines 4..8). The coverage matrix and fault matrix are referenced for each test case to increment the counters (Lines 9..16). It is possible that no test cases hit the coverage requirement, in which case the algorithm moves forward to the next one (Line 17). The counters are then used to calculate the fault correlation number (Line 18), and the maximum probability of detecting the fault is potentially updated (Line 19). After all coverage requirements and faults are evaluated, the average probability of detecting each fault is calculated (Line 22).

This chapter has defined the traditional metrics for percentage size reduction and percentage fault detection reduction, as well as a novel metric based on the average probability of detecting each fault in a reduced suite. In the next chapter, these metrics will be applied to the results of a set of experiments to compare the performance of different coverage criteria when used in test suite reduction.

Chapter 6: Experiments

Five experiments were performed to evaluate the effectiveness of call-stack-based test suite reduction, each to address a specific research question. This chapter will discuss the conduct of the experiments and their results.

6.1. Research Questions

Five research questions are posed in the following subsections. Research Questions 1 through 4 (Q1-Q4) are addressed using experiments that capture the percentage size reduction and percentage fault detection reduction as given in Equations (8) and (9). Since these experiments deal with a fairly small number of discrete faults (as will be seen in Section 6.2), averages of these quantities were taken over large numbers of suites. Research Question 5 (Q5) will be answered using the average probability of detecting each fault, calculated as given in Equation (16). To further explore Q5, analyses based on the number of faults always found in coverage-equivalent reduced test suites will also be performed.

6.1.1. Research Question 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?

For call stack coverage to be compelling as a criterion for test suite reduction, it should provide new and useful characteristics for size reduction and fault detection effectiveness when compared to previously existing criteria. This research compares coverage techniques using experiments involving both conventional, batch-oriented

applications and modern GUI applications. (Subject applications are discussed in detail in Section 6.2.) Method coverage (M) is compared to call stack coverage (CS) for all subjects. For GUI subject applications, event (E) and event-interaction (E2) coverage are added. (E1 and E2 do not apply to non-GUI applications.) Additionally, line coverage is used where readily available, specifically, in the Java-based subjects.

Event-based coverage [29] has been developed specifically for applications where test cases can be defined as sequences of events, and as such it is particularly suited to GUI applications. Examples of events in GUI applications include button clicks, menu selections, and keystrokes. In this work, reduction techniques based on two different event sequence lengths are considered. In E1, each event in isolation is a coverage requirement to be covered by any reduced test suite, and in E2, coverage requirements are made up of pairs of events.

Lines, methods, and edges are program elements measured in well-studied, traditional test coverage techniques. In *line coverage*, the coverage of each source code line induced by test execution against a given subject application is measured. From this, it is possible to define reduction technique L, in which reduced test suites must obtain the same line coverage as their full counterparts. *Method coverage* is used to reduce test suites in the M technique. In M, each method appearing in the full test suite must also appear in the corresponding reduced suite. This information is derivable from call stack coverage data and does incorporate coverage of libraries for Java programs.

6.1.2. Research Question Q2

How does 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 average than another technique. Q2, therefore, removes size as an independent variable. Here, it is investigated whether test suites created by call stack reduction preserve more fault-detecting ability than randomly reduced suites of the same size, as well as line, event, and method-reduced suites (as applicable) augmented with additional random test cases to make them the same size.

6.1.3. Research Question Q3

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

A significant advantage of the call stack criterion is its ability to capture interesting behavior from platform libraries without necessarily requiring invasive instrumentation of those libraries. For applications that make extensive use of libraries such as the Java 2 Software Development Kit (SDK), it is informative to evaluate the impact of including library routines in method and call stack reduction on size reduction and fault detection reduction. This can be accomplished by reducing test suites using coverage information that includes library methods, reducing the same suites while excluding library methods, and comparing the resulting reductions in size and fault detection.

6.1.4. Research Question Q4

Does call-stack-based test suite reduction perform differently between conventional and event-driven applications?

Key to the idea of using call stacks is the notion of context. That notion is very strong in GUI applications, where multiple degrees of freedom in the interface allow events to be executed from many different states. Context is often not as important a factor in conventional or batch-oriented applications, which could cause call-stack-based test suite reduction to be less desirable in such scenarios. Thus, to see if call-stack-based test suite reduction is sensitive to the style of application, its behavior was compared between non-event-driven, non-GUI applications to that observed for event-driven GUI applications.

6.1.5. Research Question Q5

Are certain types of coverage requirements more often associated with faults?

If a specific coverage requirement is covered primarily by fault-revealing test cases, this intuitively provides strong evidence that the coverage requirement in question is related to a fault. Moreover, if a coverage requirement is *only* hit by fault-revealing test cases, no coverage-preserving test suite reduction technique can possibly lose that fault. So in practice, it would be useful to identify and select a coverage technique that can be expected to maximize, on average, the number of such coverage requirements. This leads to insight into which coverage criteria are best to use in coverage-preserving test suite reduction algorithms: If coverage criteria differ in how strongly their coverage requirements are associated with *known* faults, this correlation will hopefully generalize to unknown faults and faults in different

applications when applied in practice. The fault detection probability metric defined in Section 5.3 is suitable for this analysis.

6.1.6. Overview of Experiments

To answer these research questions, five experiments were designed and are presented in the remainder of the chapter. In Experiment 1, call-stack-based reduction was compared with event, event-interaction, line, and method-based reduction for GUI subjects, and conventional criteria for non-GUI subjects. Experiment 2 compared call stack reduction to randomly selected and augmented line, event, and method-reduced suites of the same size. In Experiment 3, method coverage and call stack coverage excluding information about library methods were considered. Experiment 4 explores any differences observed in the effectiveness of call-stack-based reduction between conventional and event-driven GUI applications. And Experiment 5 relates coverage requirements to fault-revealing test cases for various types of coverage, using the new metric based on the *average probability of detecting each fault*.

6.2. Subject Applications

This research requires experimental subject applications which have a universe of test cases and a set of known faults. Additionally, the absolute counts of test cases and faults for each application must be sufficiently large to support calculations based on reduction in suite size and fault detection. As is often the case in empirical studies in software testing, very few appropriate subject applications are generally available. In various experiments, this research will utilize five subject

applications that have sufficient fault and test case information and display characteristics of interest in terms of programming style, source language, and execution style. Characteristics of these applications are listed in Table 1, and Table 2 shows their test case and fault information. These applications are discussed in more detail in the following sections.

Application	**Source Language**	**Execution Style**	**Programming Style**
TerpPaint	Java	Event-Driven (GUI)	Object-Oriented
TerpWord	Java	Event-Driven (GUI)	Object-Oriented
TerpSpreadsheet	Java	Event-Driven (GUI)	Object-Oriented
Space	C	Conventional	Procedural
nanoxml	Java	Conventional	Object-Oriented

Table 1: Subject Applications Characteristics

Application	**TerpPaint (TP)**	**TerpWord (TW)**	**TerpSpreadsheet (TS)**	**nanoxml**	**space**
Test Universe Size	1500	1000	1000	216	13585
# Detectable Faults (Versions)	43	18	101	9	34

Table 2: Subject Application Test Cases and Faults

6.2.1. TerpOffice

Object-oriented, event-driven GUI applications are taken from the TerpOffice Suite [30] to be subjects for the following experiments. 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). Each TerpOffice application is associated with a

large universe of test cases generated using the event flow criterion [26] and a set of seeded mutation faults. Each application comes with a set of versions each containing a single known fault and fault detection matrix for each test case.

6.2.2. Space

The well-studied `space` application [37] is used as a conventional, non-GUI subject application. `Space` is an antenna-steering system developed by the European Space Agency commonly used in studies of test suite reduction, test case prioritization, and regression test selection. It is written in C in the procedural style and executes sequentially. The version used in this research comes with 13,585 test cases and 34 known faults. Line coverage information for `space` was not available and therefore not used in subsequent experiments.

6.2.3. nanoxml

Nanoxml is a small XML parser which, like `space`, is a conventional non-GUI application but, like TerpOffice, is written in Java and makes use of the Java libraries. It was obtained via the Software-artifact Infrastructure Repository (SIR) hosted at the University of Nebraska [7]. Nanoxml exists in multiple versions to support different types of experiments. A single version of nanoxml was used in this research which has 216 test cases and nine known faults.[2] Unlike the other subject applications, most of nanoxml's faults are detected by a large percentage of the test

[2] With nine, nanoxml has the smallest number of known faults of any of the subject applications used in this research. Small numbers of faults present difficulties for test suite reduction research. In the case of nanoxml, an individual fault missed by a reduced test suite increases fault detection reduction by 11.1%. Other applications from SIR were rejected because they had even fewer known faults.

cases in its universe. As a result, certain reduction techniques (such as those that involve random test case selection) would be expected to perform relatively better.

6.3. Experimental Procedure

Figure 5 shows a general procedure used for conducting test suite reduction experiments which is used in this research. Ovals represent tools/processes; boxes represent experimental artifacts/results; hexagons represent calculated metrics. For each subject application, the process begins with a pool of test cases, a set of known faults, and a fault matrix, *i.e.*, information on which test cases detect which faults. The version of the subject application itself used in the coverage collection process contains none of the known faults and is therefore deemed to be "fault-free". This approach ensures that a complete set of coverage data may be collected without the collection process being confounded by faulty behavior. Subsequent use of this data as a coverage baseline for test suite reduction simulates the realistic situation where faults are introduced over time during the development process and found via regression testing. After coverage data is collected, the following steps are performed:

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 (non-reduced) test suite, calculate the set of faults 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 it detects.

6. Compute the percentage size reduction and percentage fault detection reduction.

This approach is discussed in more detail in the sections that discuss experiments where the procedure was applied.

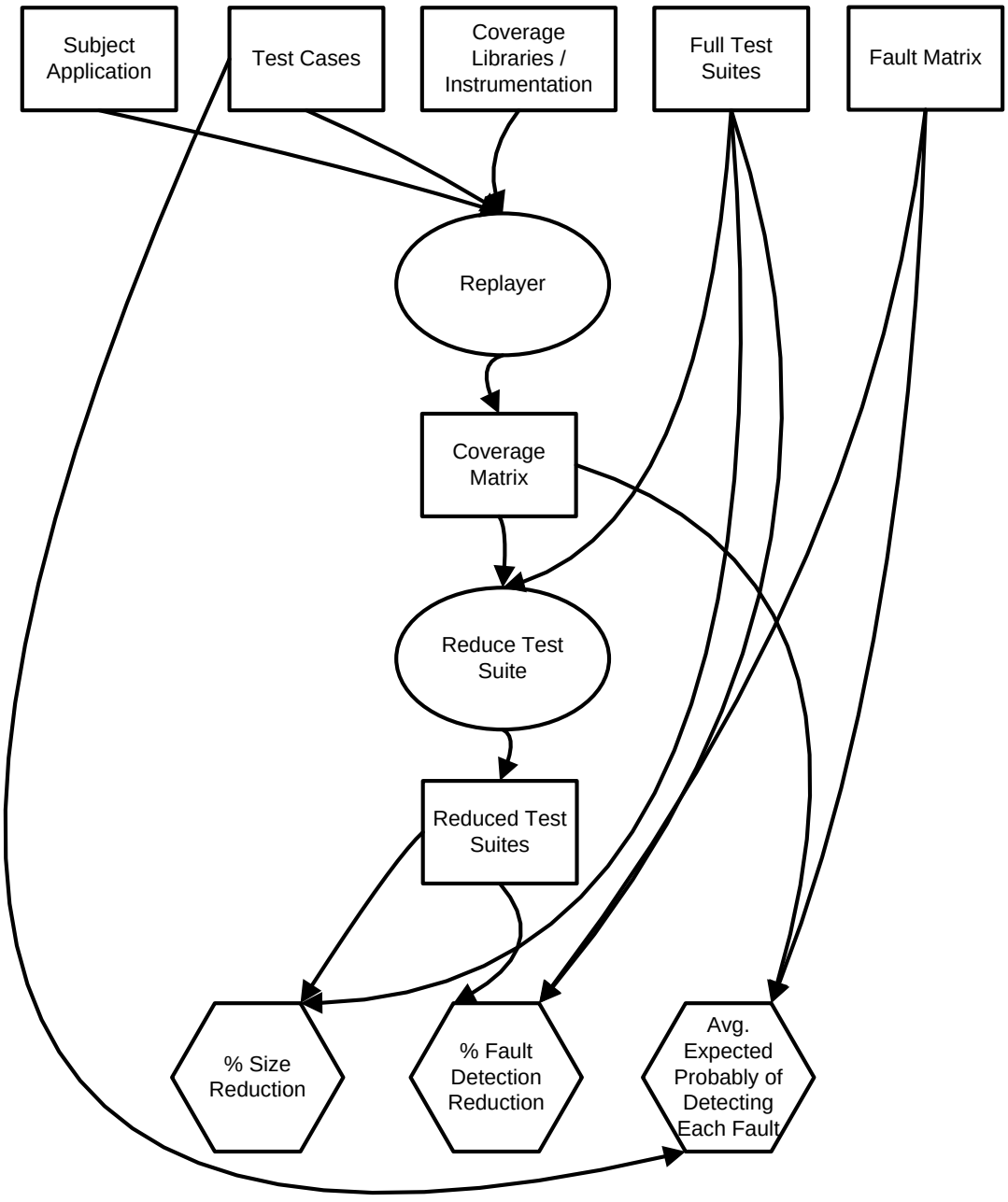

Figure 5: Experimentation Procedure

6.4. Threats to Validity

6.4.1. Threats to External Validity

Threats to external validity are factors that may impact the ability to generalize the results of this research to other situations. The main threat to external validity in these experiments is the small sample size. This research conducts test suite reduction experiments on a total of only five programs, which were chosen for their availability and the fact that they came with a sufficient number of test cases and known faults to support experimentation. Three of these programs were constructed by undergraduate students, one by a governmental entity, and one is open-source. These applications therefore may not be representative of the broader population of programs. An experiment that would be more readily generalized would include additional programs of different sizes and from different domains. Additionally, one would expect the effectiveness of the call stack reduction 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. Three of the subject applications used in this research 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.) Also, this research, like much related work in the areas of test suite reduction, prioritization, and regression test

selection, performs experiments and analyses involving known faults. This type of research assumes that the known faults are representative of the set of all faults which may appear in the subject applications, which may or may not hold in practice. This is an even larger threat when using subject applications with a very small number of known faults, such as nanoxml, which only includes nine. 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 that are adequate for a variety of coverage criteria.

6.4.2. Threats to Construct Validity

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 [[36][42]]. 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 did not include coverage of the underlying library code, in contrast to the approach taken for method coverage. Including line coverage of libraries may alter the performance of line-based test suite reduction relative to the other coverage criteria.

6.4.3. Threats to Internal Validity

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 the results and the conclusions drawn from them. These threats have been controlled for by testing the tools and the data quality.

6.5. Data Collection Step

6.5.1. Collection Process

Coverage data from each subject application was collected before beginning the experiments. The data gathered during this step allowed for the creation of any number of test suites composed of the previously executed test cases. In each such test suite, the set of unique coverage requirements and faults detected by the suite are known without further execution of the subject applications. Hence, it was not necessary to run each test suite against each version of the applications under study. This simulation approach is similar to one used by Frankl et al. [11] to evaluate adequacy criteria and test effectiveness.

For the TerpOffice applications, the JavaGUIReplayer application [30] (shown as "Replayer" in Figure 5) was used to execute each test case in each test pool against the fault-free versions of the subject programs. Initially this process was used with JavaCCTAgent to collect the unique call stacks generated by each test case. This process was then repeated to collect line coverage using jcoverage [19] or Cobertura [5] as the instrumentation tool. Method coverage was derived from the call stack coverage data. Because the test cases for the GUI subject applications were event-based, their event coverage was known *a priori*. Coverage statistics aggregated over the entire test pool for each GUI application appear in Table 3. For each subject application, the first two rows of Table 3 list the total number of unique call stacks and methods (including library methods, not limited to 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 executable lines, classes, and methods comprising each application, as determined by the jcoverage instrumentation tool.

	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
# Executable Lines	No	11803	9917	5381
# Classes	No	330	197	135
# Methods	No	1253	1380	746

Table 3: GUI Subjects' Static and Dynamic Program Elements

A similar process was used to collect data for the conventional applications, nanoxml and `space`. As these applications are not event-driven, event coverage does not apply. As mentioned earlier in Section 6.2.2, line coverage was not addressed for `space`, and coverage of the C libraries used by `space` is subject to the limitations of the Detours-based implementation discussed in Section 4.1.2. Coverage statistics for the conventional applications appears in Table 4.

	Includes Library Data?	**nanoxml**	**Space**
# Call Stacks Observed	Yes	6617	453
# Methods Observed	Yes	1126	143
# Executable Lines	No	3012	6218
# Classes	No	25	N/A
# Methods	No	232	123

Table 4: Conventional Subjects' Static and Dynamic Program Elements

6.5.2. Coverage of Library Elements

As noted in Section 6.1, the instrumentation process for call stack coverage of the Java subjects used in this research incorporates the coverage of the supporting Java libraries induced by test case execution. Because the raw call stack coverage data was used as the basis for method coverage, the method coverage approach also includes Java library methods. However, because it was not feasible to instrument the entire Java SDK for line coverage, line coverage data is based solely on the subject application source. Because of this, between the two approaches M and L, it is possible (and in fact the case) that tests may cover more methods than lines.

6.6. Reduction Approach

Before reducing a test suite, the individual test case coverage information from Section 6.6 is used to calculate the full set of unique maximum-depth 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 the situation where a maximum-depth call stack from one test case is not maximum-depth in another must be considered. 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 maximum-depth in a test suite containing both *tc1* and *tc2*. Two separate approaches were used in this research. For the `space` application, this issue was addressed by computing substack relationships between each pair of unique maximum-depth call stacks globally, across the suite. In the example, this would lead to a selection of just *tc1*, because it covers both stacks *c1* and *c2*. However, it was observed that computing the substack relationships across an entire test suite with hundreds of thousands of unique (and deep) call stacks as in the Java-based applications is very computationally expensive. Therefore, the experiments using the Java applications take 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 maximum-depth stacks are treated across threads as discussed in Section 3.1. So in the example, 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 better

runtime performance of the reduction process. The differences in reduction across the two approaches are expected to be very minor in practice, but future work may quantify the delta in size reduction.

After merging the unique maximum-depth call stacks from each test case in a given test suite, the *ReduceTestSuite* heuristic [13] is applied to compute the reduced test suite. Finally, the desired metrics are calculated based on the reduced suite.

6.7. Experiment 1: Comparing Coverage-Based Reduction

The goal of Experiment 1 was to reduce randomly generated test suites of various sizes based on call stack coverage (CS) and the other coverage criteria under study: event (E1), event-interaction (E2), line (L), and method (M) as applicable for the Java-based subjects, and method (M) for `space`. Due to the differences in test universe size across the subject applications, different suite sizes were evaluated. The sizes by application are listed in Table 5. Test suites were reduced based on each of the evaluated criteria and compared in terms of the percentage size reduction and percentage fault detection reduction metrics.

Application	**Suite Sizes Evaluated**	**Number of Suites Per Size**
TerpPaint, TerpWord, TerpSpreadsheet	50, 100, 150, 200, 250, 300, 350. 400	25
Nanoxml	20, 40, 60, 80, 100, 120, 140, 160	100
Space	50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800, 850, 900, 950, 1000	50

Table 5: Random Suite Sizes Tested by Subject Application

6.7.1. Size Reduction

Percentage size reduction results for the five subject applications for each applicable reduction approach appear in Figures 6 to 10. (The SM and SCS approaches will be discussed in conjunction with Experiment 3, Section 6.9.)

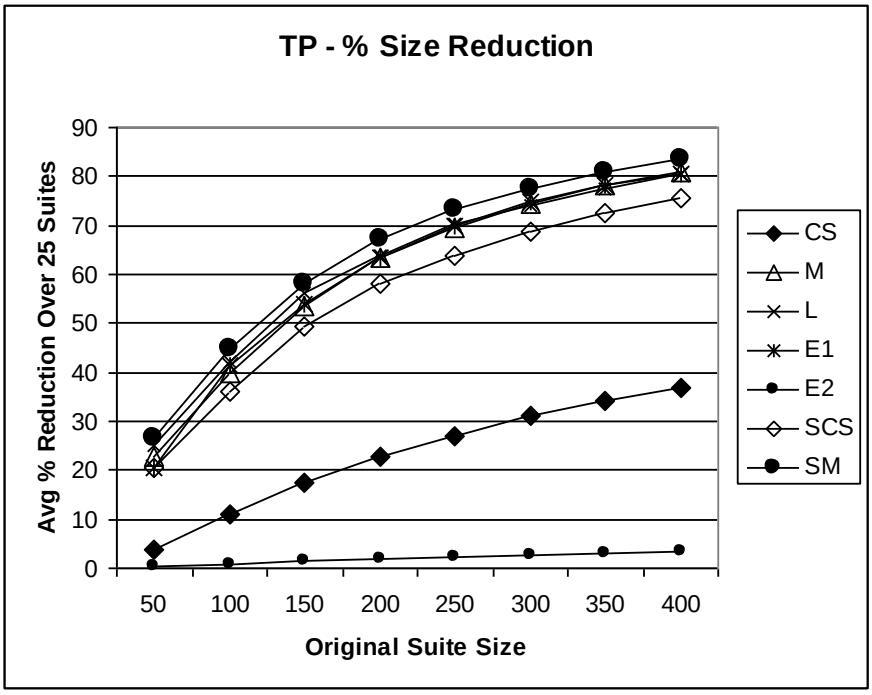

Figure 6: TP Percentage Size Reduction

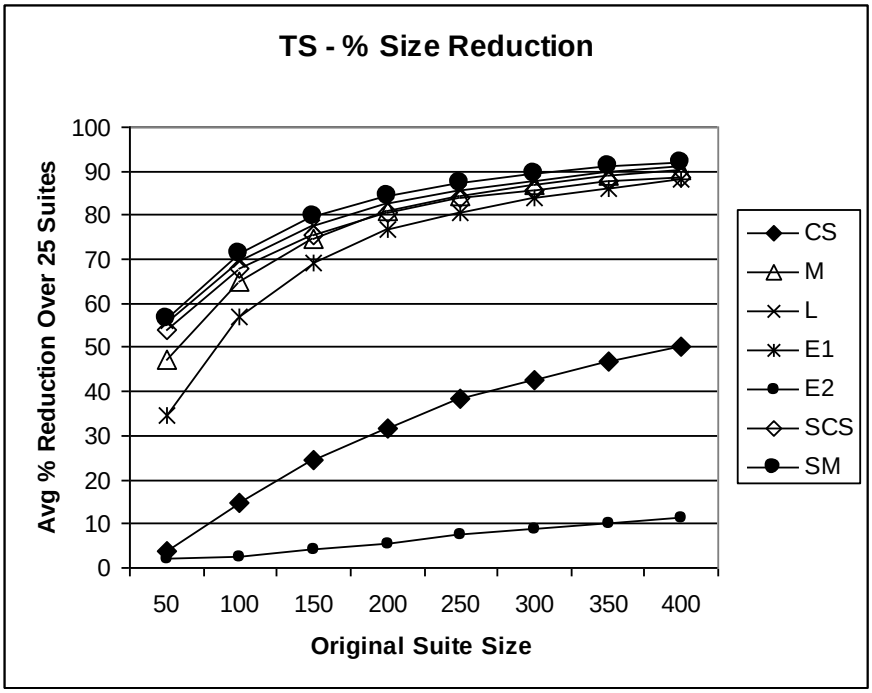

Figure 7: TS Percentage Size Reduction

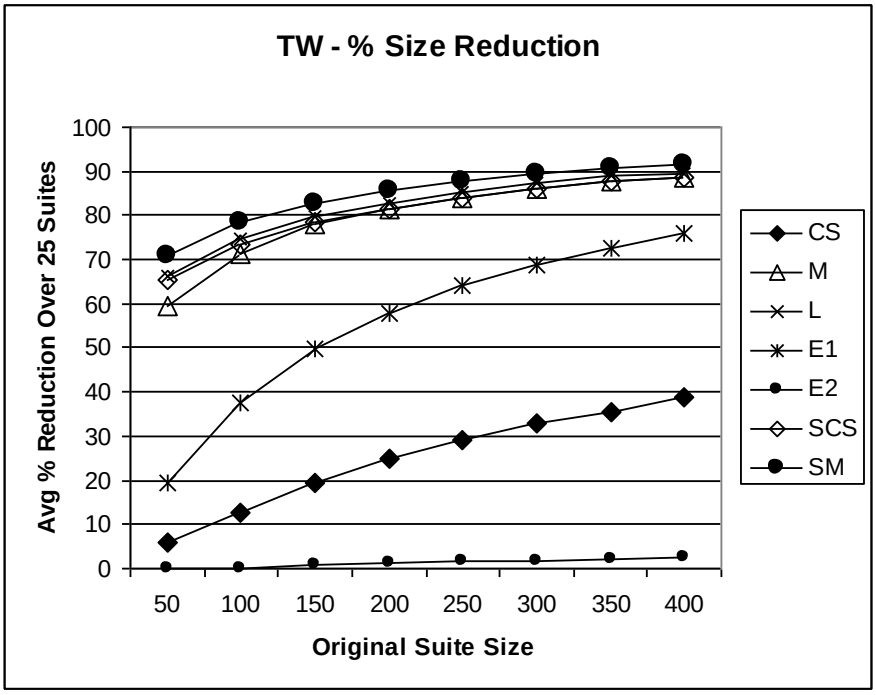

Figure 8: TW Percentage Size Reduction

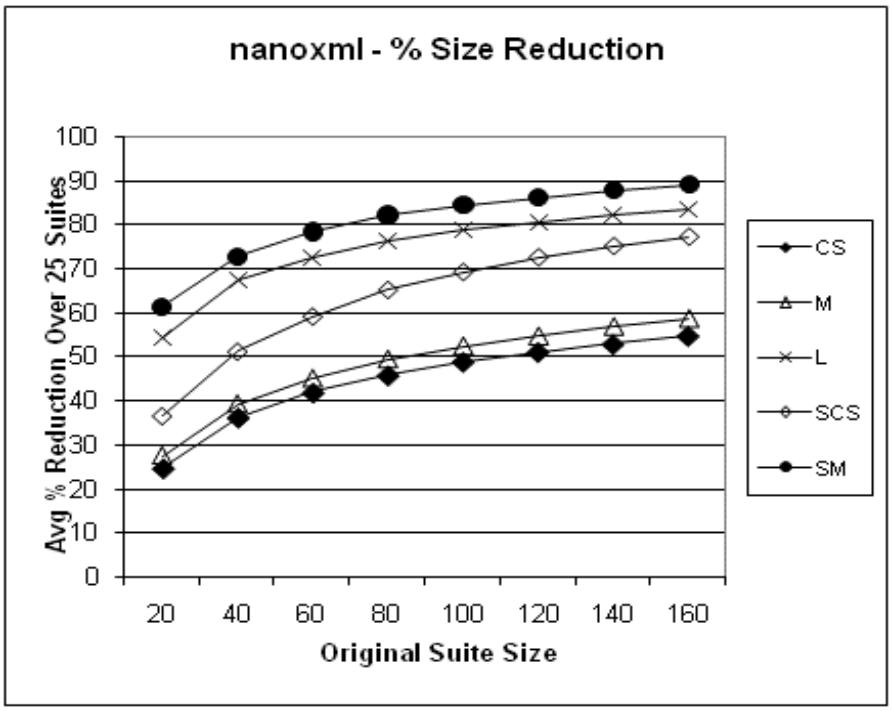

Figure 9: Nanoxml Percentage Size Reduction

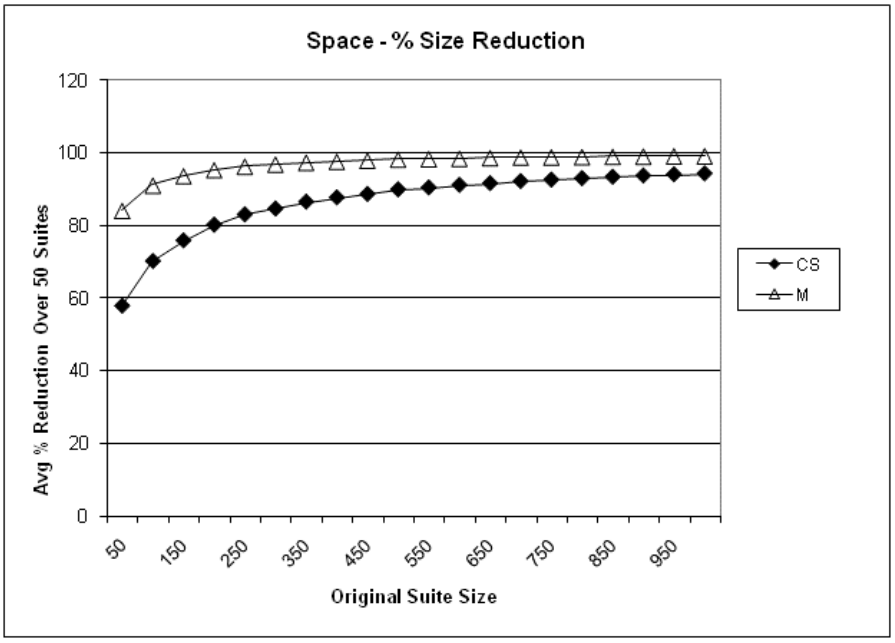

Figure 10: Space Percentage Size Reduction

Similar behavior in suite size reduction is observed for all three GUI subjects. 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 still notably larger 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. For the non-GUI subject applications, the CS approach still results in less size reduction than the comparison techniques. However, it is much closer than in the GUI applications. This phenomenon will be considered in more detail in Section 6.11.

To evaluate the statistical significance of differences between CS and the other techniques seen in Figures 6 through 10, 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 percentage size reduction" and means of each of the other techniques. The results appear in Table 6. Since all the p-values for percentage size reduction are below 0.05, the null hypothesis is rejected and the alternative hypothesis, *i.e.*, there is a statistically significant difference between the means of CS and the other techniques, is accepted.

CS vs.	**% Size Reduction p-Value**				
	TP	**TS**	**TW**	**nanoxml**	**Space**
M	7.84E-06	6.04E-09	3.52E-10	6.23E-08	2.06E-7
L	3.02E-06	2.9E-08	1.29E-09	1.55E-12	--
E1	1.13E-05	4.59E-08	1.36E-05	--	--
E2	0.000823	0.000932	0.000414	--	--
SCS	7.13E-06	4.85E-08	1.54E-09	2.61E-06	--
SM	2.95E-06	2.96E-08	3.61E-09	1.07E-12	--

Table 6: Paired-t Testing for Size Reduction of CS vs. Other Techniques

6.7.2. Fault Detection Reduction

Percentage fault detection reduction results for the five subject applications appear in Figures 11 through 15. (The RAND, E1A, LA, MA, SCS, and SM techniques will be discussed in subsequent experiments 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.

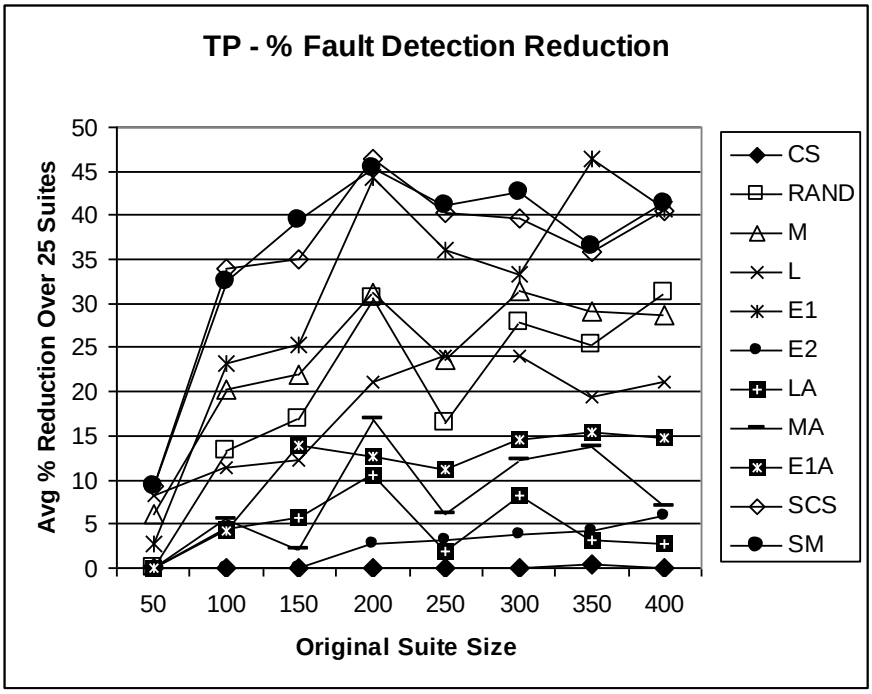

Figure 11: TP Fault Detection Reduction

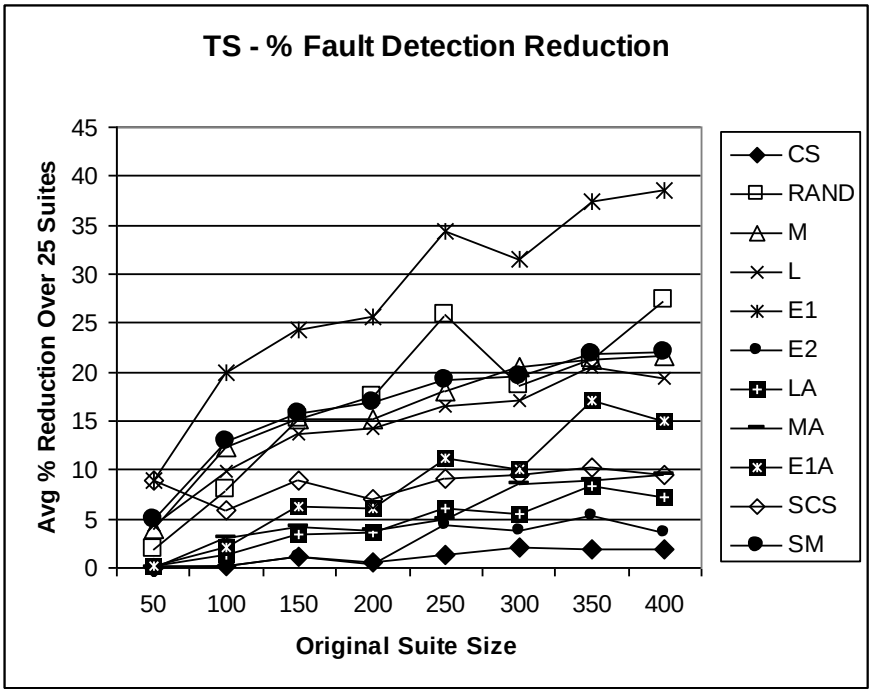

Figure 12: TS Fault Detection Reduction

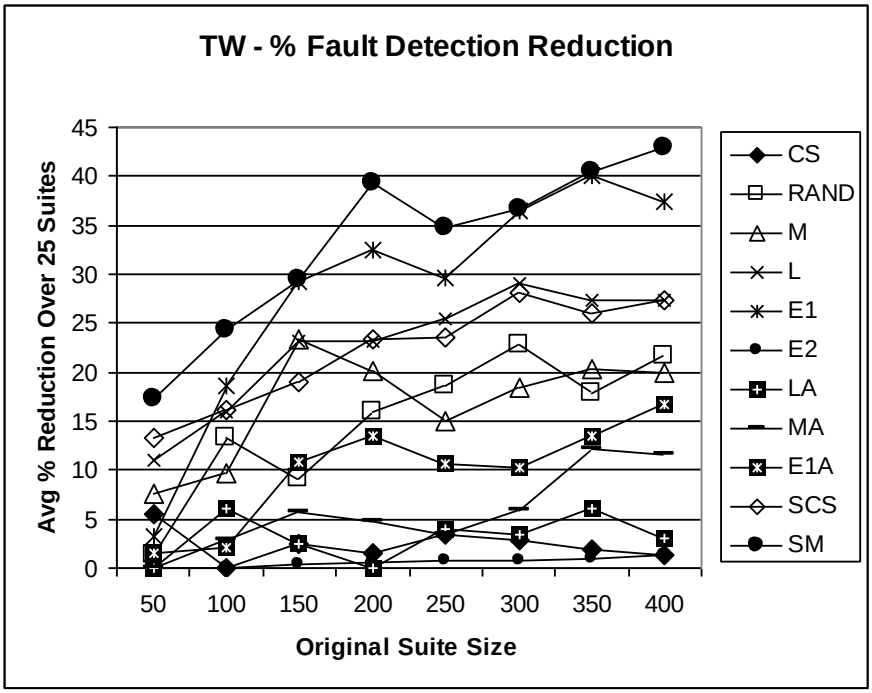

Figure 13: TW Fault Detection Reduction

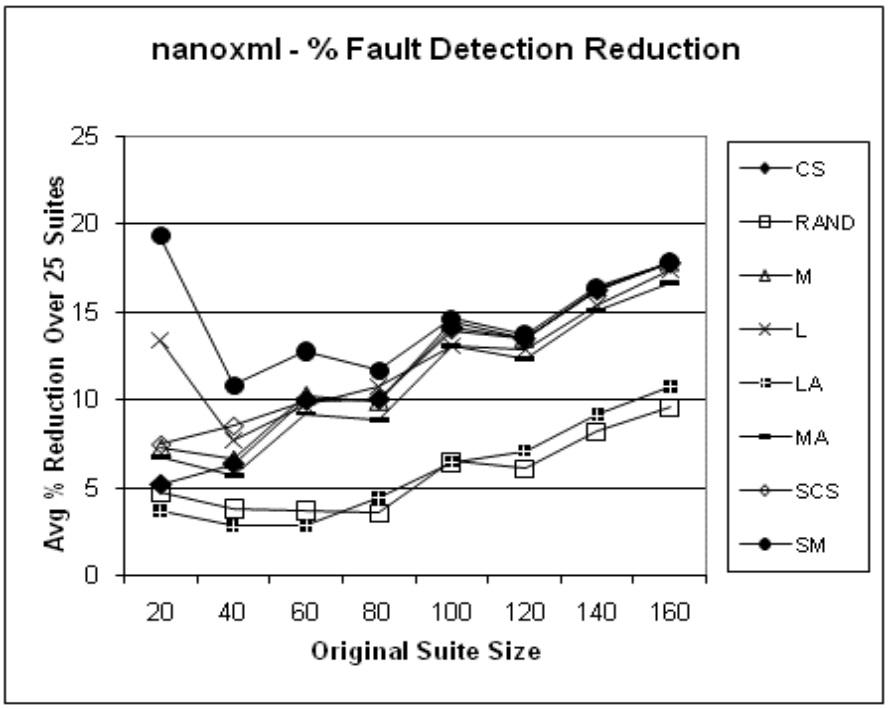

Figure 14: Nanoxml Fault Detection Reduction

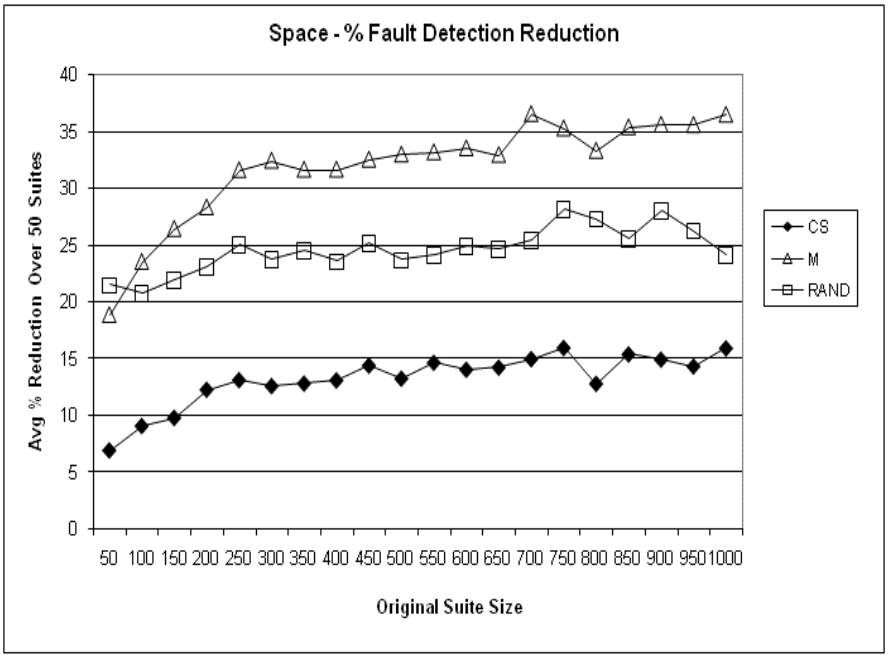

Figure 15: Space Fault Detection Reduction

As with percentage size reduction, there is no clear difference between M and L in the GUI subjects (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 these experiments.

Call stack coverage also performs relatively well in the non-GUI `space` application, with fault detection reduction less than half of that observed for method-based reduction. For nanoxml, however, results are less clear. Seven of the nine faults in nanoxml are detected by a large number of test cases, which allows the techniques that utilize random test case selection (RAND, LA, and MA) to perform relatively well. The CS technique is virtually indistinguishable from M and L. For both conventional applications, the magnitude of percent fault detection reduction is notably higher than for the GUI subjects (up to 17% versus less than 5%). 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 modern applications.

To evaluate the statistical significance of the difference of means between CS and the other reduction techniques as seen in Figures 11 through 15, 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 Table 7. For the TerpOffice applications and space, since all p-values of M, L, E1, and E2 for percentage fault detection reduction are below 0.05, the null hypothesis is rejected and the alternative hypothesis, *i.e.*, there is a statistically significant difference between the means of CS and these techniques for

all subject applications, is accepted. For nanoxml, the null hypothesis cannot be rejected, and in fact, it is observed that the best techniques are RAND and LA (discussed in Experiment 2).

CS vs.	% Fault Detection Reduction p-Value				
	TP	**TS**	**TW**	**nanoxml**	**Space**
RAND	0.001041	0.000803	0.002916	0.000665	2.67E-18
M	8.48E-05	8.13E-05	0.000353	***0.199924***	6.98E-19
L	8.07E-05	7.02E-05	7.26E-05	***0.457429***	--
E1	0.000426	8.9E-05	0.000792	--	--
E2	0.016876	0.039215	0.025051	--	--
LA	0.007803	0.002918	***0.553965***	0.000133	--
MA	0.006307	0.002236	***0.10448***	***0.073261***	--
E1A	0.000976	0.005401	0.010153	--	--
SCS	4.63E-05	1.11E-07	3.89E-05	***0.226754***	--
SM	4.78E-05	4.68E-05	4.35E-05	***0.122060***	--

Table 7: Paired-t Testing for Fault Detection Reduction of CS vs. Other Techniques (Bold Values Not Statistically Significant at the 0.05 Level)

In summary, this experiment finds that call-stack-based reduction of test suites for event-driven applications results in measurable size reduction and extremely low fault detection reduction compared to other techniques. For conventional applications, call-stack-based test suite reduction provides an effective tradeoff in size reduction versus fault detection reduction in one of two subject applications. This result answers research question Q1. Additionally, the data collected in Experiment 1 will be leveraged to answer additional research questions in the subsequent sections.

6.8. Experiment 2: Controlling for Size of Reduced Suite

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 call stack coverage may have been preserving more fault detection capability 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 Figures 11 through 15) and compared with respect to their fault detection effectiveness. Also, the reduced suites resulting from L, M, and E1 were randomly augmented 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" or "augmented" techniques are the LA, MA, and E1A techniques, respectively, in Figures 11 through 15

Referring back to those figures, in the GUI applications 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. As per Table 7, for two of the three GUI subjects, CS shows significantly better percentage fault detection reduction. For TW, the LA and MA techniques are not statistically distinguishable from CS. For the conventional application nanoxml, Table 7 shows statistical evidence that CS is *not* an improvement over same-sized suites created using other approaches. In fact, RAND, LA, and MA also appear to perform better than the other coverage-based approaches, L and M. Thus it appears that reduced test suite size is a more important influence on fault detection than coverage for this application, possibly because most of nanoxml's faults are detected by a large number of test cases in the universe.

Considering that the suite sizes from RAND, E1A, LA, and MA are equal to those of CS, this research concludes that in most cases, call stack coverage contains valuable information that preserves fault detecting ability of test suites under reduction in modern GUI subject applications. There is no evidence that this is the case in conventional, non-GUI subjects, but further research is needed to clarify this point. This result addresses research question Q2.

6.9. Experiment 3: Omitting Library Methods

In research and in industrial practice, most coverage techniques are evaluated only on those coverage requirements which can be derived from first-party source code. This research hypothesizes 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, coverage information was generated for both methods and call stacks excluding methods from the Java platform libraries. (Because only 20 of 143 methods observed in space mapped to methods in the Standard C Library, this experiment was not performed for that application.) These techniques are called "SCS" (for "short" call stack) and SM (for "short" method) in Figures 6 through 15. The numbers of coverage requirements for the applications under study appear in Table 8. Because most Java 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.

Application	Observed Method Count Excluding Library Methods	Observed Call Stack Count Excluding Library Methods
TerpPaint	680	923
TerpWord	757	1780
TerpSpreadsheet	525	2653
nanoxml	137	652

Table 8: Non-Library Coverage Statistics

When test suite reduction is performed based on the "short" call stack and method coverage data, size reduction is very comparable to L, M, and E1 in all of the GUI subject applications. For the non-GUI nanoxml application, size reduction when using the "short" techniques is less comparable to L and M. As can be seen in Tables 9 and 10, size reduction relationships were found to be statistically significant except for SCS versus M in TS and TW.

SCS vs.	% Size Reduction p-Value			
	TP	**TS**	**TW**	**nanoxml**
M	1.79E-05	***0.520725***	***0.21264***	5.02E-06
L	1.75E-07	3.49E-08	2.95E-08	0.000153
E1	0.000321	0.032248	0.000448	--
E2	7.11E-05	8.13E-08	8.42E-09	--
SM	1.29E-07	3.75E-08	4.51E-06	1.73E-05

Table 9: Paired-t Testing of SCS vs. Other Techniques for % Size Reduction (Bold Values Not Statistically Significant at the 0.05 Level)

SM vs.	% Size Reduction p-Value			
	TP	**TS**	**TW**	**nanoxml**
M	4.19E-06	0.002928	0.001513	2.58E-11
L	6.8E-06	1.54E-05	5.36E-05	2.05E-08
E1	4.11E-05	0.003248	0.000284	--
E2	3.49E-05	7.24E-08	2.40E-09	--
SCS	1.29E-07	3.75E-08	4.51E-06	1.73E-05

Table 10: Paired-t Testing of SM vs. Other Techniques for % Size Reduction

Fault detection reduction displays quite a bit of variance between the applications. For TerpPaint, SCS and SM perform very comparably to the least successful reduction technique, E1. In TerpWord, SM tracks again with E1, but SCS fares better and is comparable to the line coverage based technique, L. In TerpSpreadsheet, SM is similar to L, losing around 20% of its fault detection effectiveness for larger original suite sizes. But SCS for TerpSpreadsheet does very well, losing no more than 10% fault detection, significantly better than M, L, E1, E2, and SM as can be seen in Table 11. For nanoxml, the only statistically significant conclusion that can be drawn is that SM leads to greater fault detection reduction than L.

SCS vs.	**% Fault Detection Reduction p-Value**			
	TP	**TS**	**TW**	**Nanoxml**
M	0.000252	0.006143	0.01079	***0.625955***
L	0.000322	0.009559	***0.305519***	***0.673211***
E1	***0.176047***	0.000728	0.047302	--
E2	3.77E-05	3.27E-06	5.83E-06	--
SM	***0.235898***	0.003313	0.000137	***0.119509***

Table 11: Paired-t Testing of SCS vs. Other Techniques for % Fault Detection Reduction (Bold Values Not Statistically Significant at the 0.05 Level)

SM vs.	**% Fault Detection Reduction p-Value**			
	TP	**TS**	**TW**	**nanoxml**
M	0.000214	***0.052797***	8.68E-05	***0.114183***
L	0.000301	0.000156	0.000168	0.014247
E1	***0.125103***	0.000277	0.023723	--
E2	3.96E-05	2.83E-05	1.19E-05	--
SCS	***0.235898***	0.003313	0.000137	***0.119509***

Table 12: Paired-t Testing of SM vs. Other Techniques for % Fault Detection Reduction (Bold Values Not Statistically Significant at the 0.05 Level)

Looking back at Table 8, 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 object-oriented programming 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 where CS performs well, in GUI applications. For those applications, results in Tables 7 and 12 indicate statistically significant differences between both SM and M, and SCS and CS. Thus, this research concludes 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. Additionally, Experiment 5 (Section 6.11) provides further evidence that consideration of library methods is valuable by shows that the techniques that include library methods have higher average probabilities of detecting each fault. This answers research question Q3.

6.10. Experiment 4: Conventional Application

Results using the conventional, non-GUI, non-event-driven subject applications in Experiments 1 through 3 indicate that call stack coverage can be less effective for test suite reduction in those applications than it is in modern GUI applications. Experiment 4 seeks to expand on these findings by reducing a different class of test suite in a conventional application, as well as determine whether call

stacks give us any insights into understanding the differences in test suite reduction between conventional and event-driven software.

This experiment makes further use of `space`. `Space` was used as the conventional application because, compared to nanoxml, it is most dissimilar from the GUI subjects in that it is written in C rather than Java and does not make substantial use of an underlying library or platform. Some pre-existing experimental artifacts and results from Rothermel *et al*. [36] were leveraged. Starting with 1000 test suites for *space* used by Rothermel *et al.* [37], each suite was reduced using call stack coverage and results compared to the edge coverage results of Rothermel *et al.* [36]. (*Edge coverage* measures traversals across each edge in a program control flow graph and is usually considered to be a relatively strong, yet practical, coverage criterion.) As in Experiment 2, call stack-reduced suites were also paired with like-sized randomly reduced suites. The results appear in Table 13. In this case, it is found that call-stack-based reduction results in slightly smaller reduced suites than edge coverage, but at the cost of over 7% additional loss in fault detection effectiveness. Call-stack-based reduction does perform far better than random reduction, however.

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

Table 13: Test Suite Reduction for *space*

Experiment 4 shows that call-stack-based test suite reduction can provide a good tradeoff between size reduction and fault detection reduction in a conventional application. However, compared to the findings of Experiments 1 through 3 for GUI applications, call stack coverage seems to be a more effective criterion for test suite reduction against modern event-driven, GUI applications than for conventional software. Note in Table 4 that for non-GUI applications, fewer call stacks are observed on a percentage basis relative to methods and lines. Although further research using a wider variety of GUI and conventional subject applications is needed, one possible explanation relates to call stacks' ability to capture the context in which a given method is invoked. GUIs tend to have more degrees of freedom, and therefore more context sensitivity, 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 (*i.e.* mouse versus keyboard), the sequence of preceding events, and the state of the program. Because each such scenario potentially results in a unique call stack, call-stack-based test suite reduction will select such test cases and, by consequence, their potentially-unique fault-detecting capability. This result addresses research question Q4.

6.11. Experiment 5: Coverage Requirements and Fault-Revealing Test Cases

6.11.1. Average Probability of Detecting Each Fault

Experiment 5 explores the relationship between coverage requirements for various criteria and test cases that reveal faults, using the newly developed *average probability of detecting each fault* metric, defined in Equation 16. The CalcFaultDetectionProbability algorithm (see Figure 4) was applied to the previously

obtained fault and coverage matrices for each Java-based application[3]. Table 14 shows the resulting average expected probability of detecting each fault after test suite reduction for each application and coverage technique, including the "short" techniques proposed in Experiment 3. The box plots in Figures 16, 17, 18, and 19 show the other key statistics for individual fault probabilities, including the minimum, maximum, median, and upper and lower quartile values.

	TP	*TS*	*TW*	*nanoxml*
E1	0.51	0.52	0.47	--
E2	0.92	0.88	0.96	--
L	0.84	0.69	0.77	1.00
M	0.80	0.69	0.72	0.81
CS	1.00	0.97	0.97	0.997
SM	0.70	0.68	0.61	0.81
SCS	0.73	0.85	0.77	0.94

Table 14: Average Expected Probability of Detecting Each Fault After Test Suite Reduction

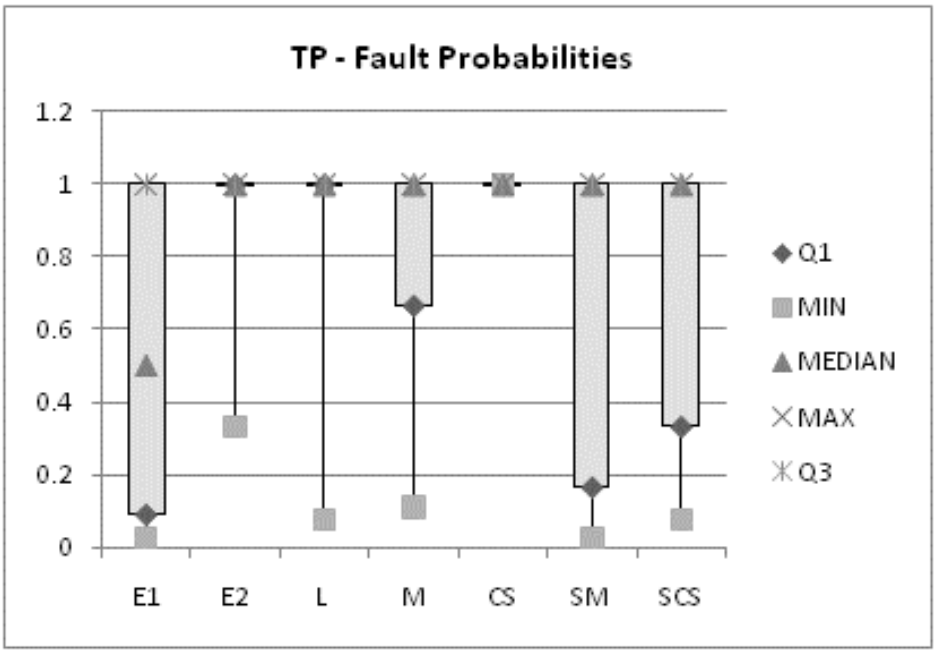

Figure 16: TP Fault Probability Statistics

[3] Space was not used in this analysis because, for reasons discussed in previous sections, its data does not enable SM, SCS, or L.

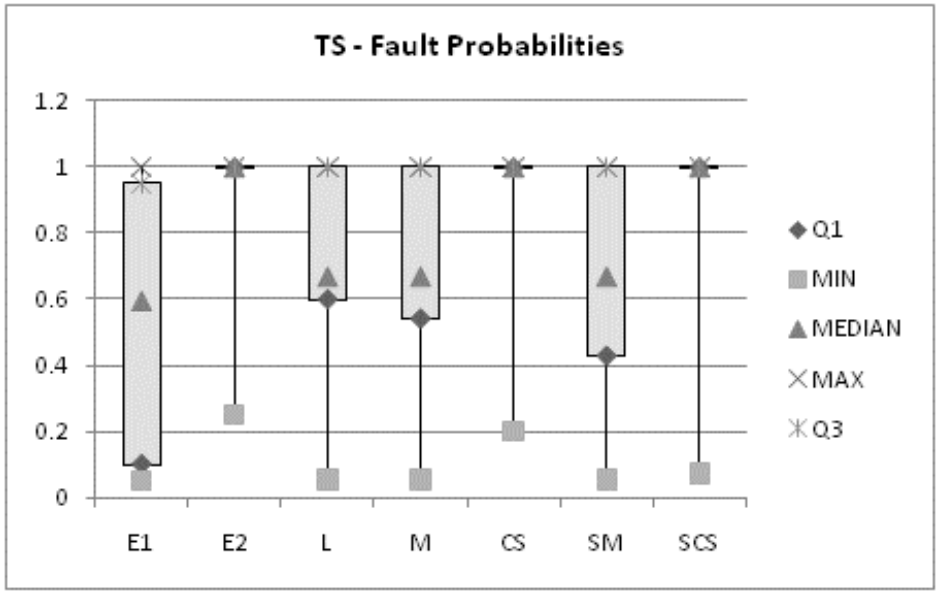

Figure 17: TS Fault Probability Statistics

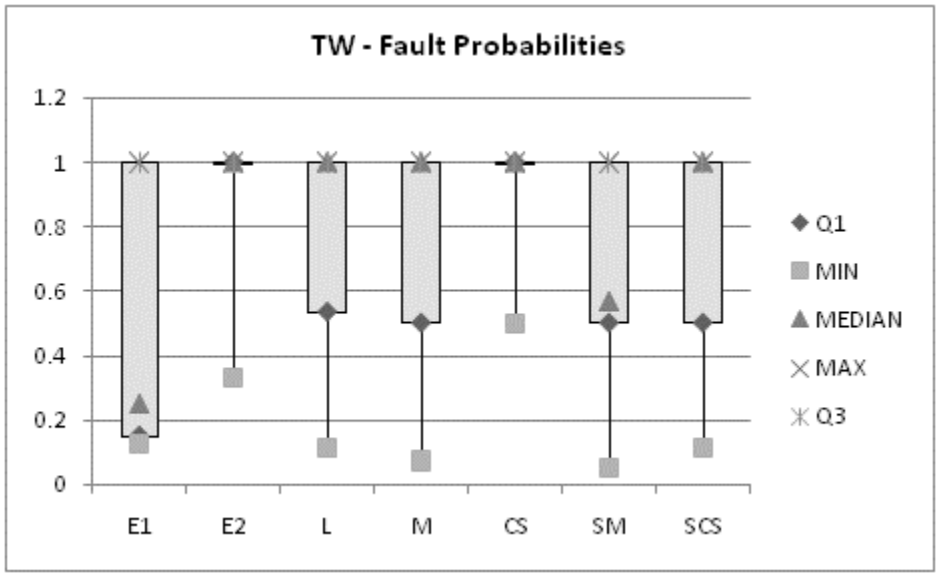

Figure 18: TW Fault Probability Statistics

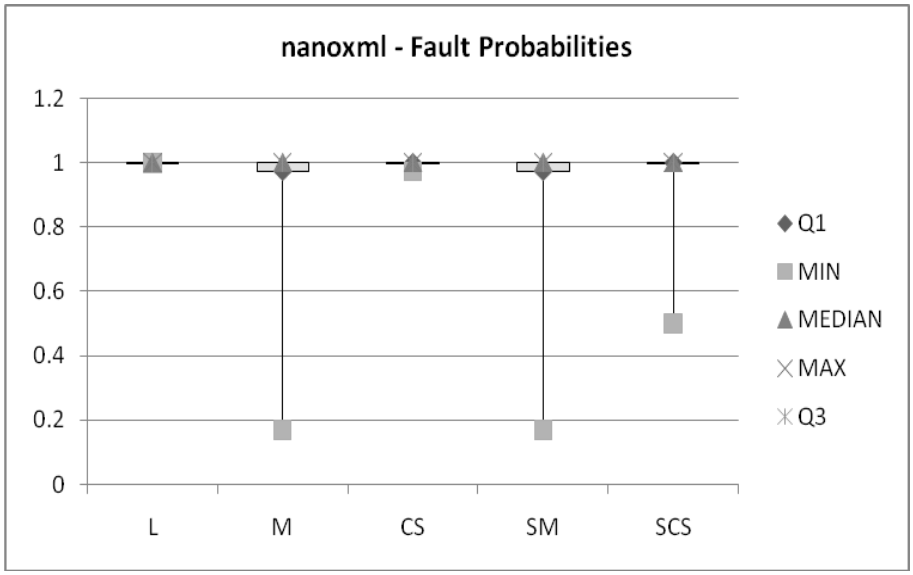

Figure 19: nanoxml Fault Probability Statistics

All of the coverage techniques perform relatively consistently across applications, the most notable exceptions being SCS's better result in TerpSpreadsheet and L's better results in TP and nanoxml. Event coverage, E1, fares the worst, while line and method coverage are comparable between 69-84% average probabilities. Event interaction coverage, E2, results in a very high average probability, but E2's usefulness in test suite reduction is limited for these subject applications and test universe as it was shown in Experiment 1 that E2 results in very large reduced suite sizes. The lower average probabilities for SM and SCS relative to M and CS, respectively, provides further evidence in favor of considering library coverage in test suite reduction. The highest average probability is achieved with the call stack coverage criterion, CS, with a 97-100% average probability of detecting each fault. This result shows quantitatively that many call stacks are highly correlated with fault-revealing test cases and therefore explains the extremely low

percentage fault detection reduction observed when using the CS technique on test suites generated randomly from this pool in Experiment 1. The maximum values in Figures 16 through 19 indicate that all coverage criteria have coverage elements that display high correlation with at least some faults. The narrow boxes for E2 and CS show that these two criteria are most effective at retaining the widest variety of faults.

6.11.2. Faults Always Detected After Reduction

When using a test suite reduction technique that preserves 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, intuition expects 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 lest coverage be lost.

This observation motivates an analysis of coverage and fault data to determine how many faults must be detected by any coverage-adequate reduced test suite on the entire test pool using the various techniques CS, SCS, M, SM, L, E1, and E2. The results of this analysis appear in Figures 20, 21, 22, and 23, where the x-axis shows the number of faults that will always be detected by any reduced suite which maintains coverage of a given criterion listed on the y-axis.

The two method-based techniques, SM and M, and L perform similarly across applications. In the conventional subject, nanoxml, L is the best technique, with all faults detected in test suites with coverage equal to that of the universe. The context-sensitive “short” call stack technique (SCS) performs comparably in TP and TW and relatively better in TS and nanoxml. Looking at the CS technique, in all but a small

handful of cases, fault-revealing test cases generate call stacks which are never observed by non-fault-revealing test cases. This phenomenon provides an explanation for the extremely low percentage fault detection reduction observed for CS in Experiment 1, lending support to the hypothesis that context information enhances coverage-based test suite reduction. Further research is needed to characterize the non-CS techniques.

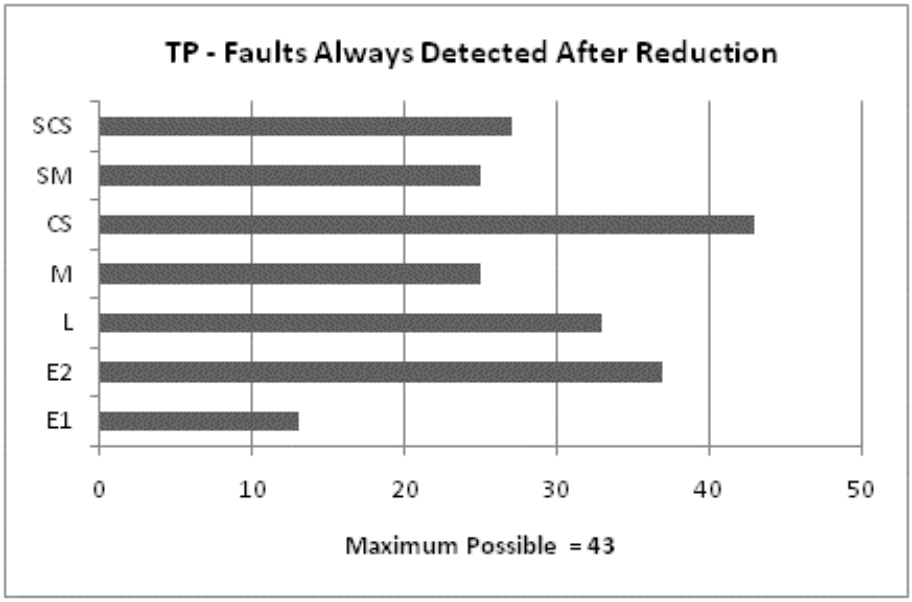

Figure 20: TP Faults Always Detected After Reduction, By Technique

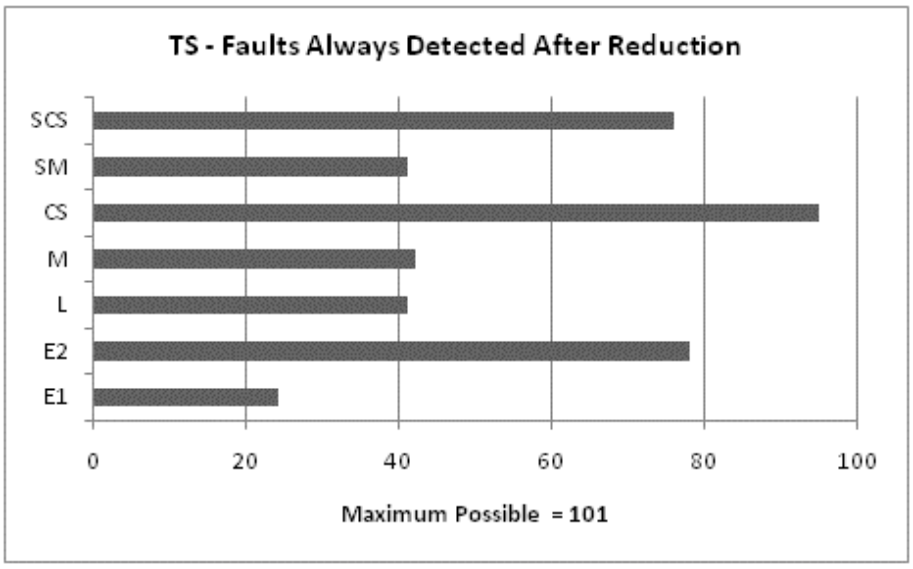

Figure 21: TS Faults Always Detected After Reduction, By Technique

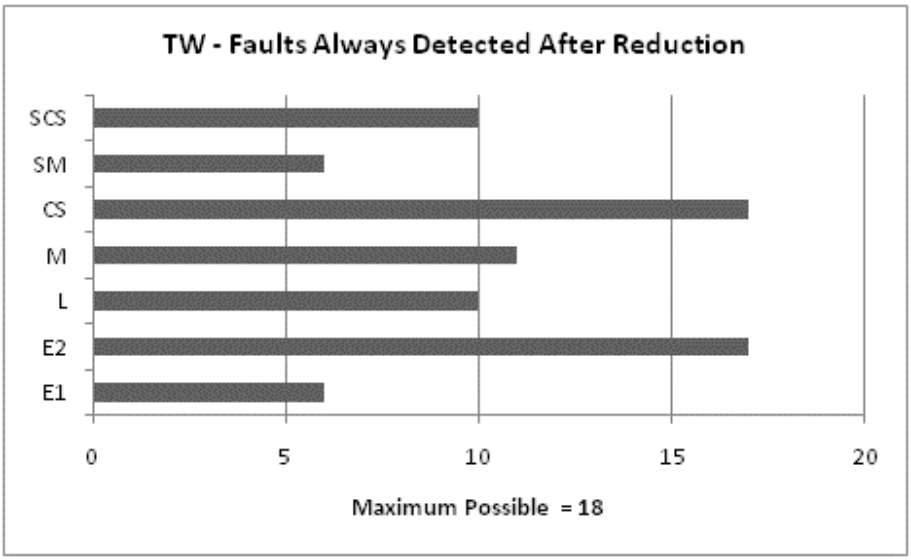

Figure 22: TW Faults Always Detected After Reduction, By Technique

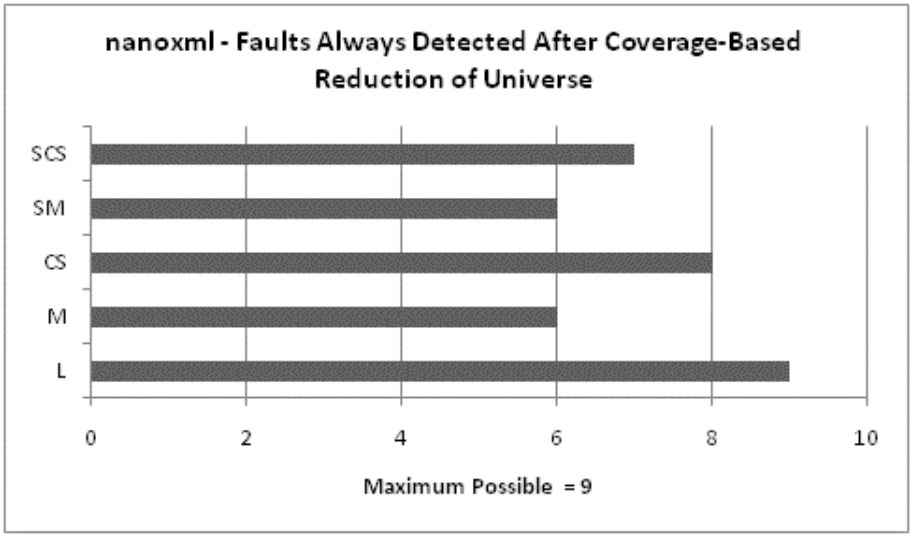

Figure 23: nanoxml Faults Always Detected After Reduction, By Technique

6.11.3. Faults Which May Be Missed After Reduction

An analysis of the faults that *can* be missed by each technique as indicated by the average probability of detecting each fault was performed, with each fault characterized by its *difficulty*. Wong *et al*. define four quartiles of faults, Quartile-I, II, III, and IV, which can be detected by [0-25)%, [25-50)%, [50-75)%, and [75-100]%, respectively, of the test cases in the test pool [42] However, by these standards, all of the known TerpOffice and nanoxml faults are "difficult" because they all fall into the low end of Quartile-I, with the median percentage of detecting cases ranging from 0.13% for TerpPaint to 11% for nanoxml. Thus, this research instead characterizes faults into three buckets based on how many test cases detect them: Hard (1-2 detecting cases), Medium (3-5 detecting cases), and Easy (6 or more detecting cases). The counts of cases per bucket were defined to give a reasonable

distribution of faults into each bucket for each application. Table 15 shows the distribution of faults by subject application.

Fault Class	*TP*	*TS*	*TW*	*nanoxml*
Easy	7	37	5	7
Medium	3	28	3	0
Hard	33	36	10	2

Table 15: Fault Difficulties

For each subject application and coverage criterion, the faults which may be lost after coverage-preserving test suite reduction were categorized. The results of this analysis appear in Table 16.

	TP		*TS*		*TW*		*nanozml*	
E1	*Easy*	*7*	*Easy*	*26*	*Easy*	*2*	*N/A*	
	Med	*3*	*Med*	*17*	*Med*	*3*		
	Hard	*20*	*Hard*	*34*	*Hard*	*7*		
E2	*Easy*	*0*	*Easy*	*0*	*Easy*	*0*	*N/A*	
	Med	*0*	*Med*	*0*	*Med*	*0*		
	Hard	*6*	*Hard*	*23*	*Hard*	*1*		
L	*Easy*	*6*	*Easy*	*13*	*Easy*	*3*	*Easy*	*0*
	Med	*1*	*Med*	*13*	*Med*	*0*	*Med*	*0*
	Hard	*3*	*Hard*	*34*	*Hard*	*5*	*Hard*	*0*
M	*Easy*	*7*	*Easy*	*12*	*Easy*	*0*	*Easy*	*1*
	Med	*3*	*Med*	*13*	*Med*	*0*	*Med*	*0*
	Hard	*8*	*Hard*	*34*	*Hard*	*7*	*Hard*	*2*
CS	*Easy*	*0*	*Easy*	*0*	*Easy*	*0*	*Easy*	*1*
	Med	*0*	*Med*	*1*	*Med*	*0*	*Med*	*0*
	Hard	*0*	*Hard*	*5*	*Hard*	*1*	*Hard*	*0*
SM	*Easy*	*7*	*Easy*	*13*	*Easy*	*3*	*Easy*	*1*
	Med	*3*	*Med*	*13*	*Med*	*0*	*Med*	*0*
	Hard	*8*	*Hard*	*34*	*Hard*	*9*	*Hard*	*2*
SCS	*Easy*	*7*	*Easy*	*2*	*Easy*	*1*	*Easy*	*1*
	Med	*2*	*Med*	*10*	*Med*	*0*	*Med*	*0*
	Hard	*7*	*Hard*	*13*	*Hard*	*7*	*Hard*	*1*

Table 16: Faults with No Coverage Requirements Unique to Detecting Test Cases by Criterion and Difficulty

No clear conclusions can be drawn for nanoxml, possibly because it has a very small number of faults (9) which are relatively easy to find compared to the TerpOffice applications. For the TerpOffice applications, the CS and E2 techniques, which only have a handful of faults overall that are not necessarily detected after reduction, show a distinct tendency for those faults to fall into the "Medium" and "Hard" difficulty buckets. For the other techniques, a trend is only visible for one of the three applications (specifically, TS). This analysis suggests that fault detection reduction in coverage-adequate reduced test suites may only be related to fault difficulty for certain coverage criteria.

6.11.4. Combining Coverage Criteria

Looking at the unique coverage requirement counts for individual faults, a number of cases were observed where Fault A is guaranteed to be detected by Technique X but not Technique Y, but Fault B for the same application is guaranteed to be detected by Technique Y and not Technique X. In other words, certain faults correlated more highly with different coverage criteria. This motivated an examination of the average probability of detecting each fault for *pairs* of criteria. Identifying effective pairs of coverage criteria is important to guide the choice of criteria to utilize in a multi-criteria test suite reduction approach such as the one proposed by Jeffrey and Gupta [20].

The following analysis assumes a test suite reduction approach that maintains coverage relative to two distinct coverage criteria. For such a coverage criteria pair, the average probability of detecting a fault is then the maximum of the individual probabilities of detecting that fault for each criterion in isolation. Data for this

analysis appears in Table 17. The pair E1+E2 is not included because E2 *subsumes* E1 – that is, an E2-adequate suite is by definition E1-adequate. The technique M+CS is omitted for the same reason, namely that CS subsumes M. Note that because M includes library coverage data and L does not, L does *not* subsume M. The "short" techniques SM and SCS are subsumed by their counterparts that include library methods, M and CS, respectively.

	TP	*TS*	*TW*	*nanoxml*
E1+L	***0.88***	***0.71***	***0.91***	--
E1+M	0.80	***0.71***	***0.82***	--
E1+CS	1.00	0.97	0.97	--
E2+M	***0.97***	***0.91***	0.96	--
E2+L	***0.96***	***0.91***	0.96	--
E2+CS	1.00	***1.00***	***1.00***	--
E1+SM	***0.77***	***0.70***	***0.76***	--
E1+SCS	***0.80***	***0.86***	***0.87***	--
E2+SM	***0.96***	***0.91***	0.96	--
E2+SCS	***0.96***	***0.98***	***0.97***	--
L+M	***0.90***	***0.70***	***0.83***	1.00
L+CS	1.00	0.97	0.97	1.00
L+SM	0.84	***0.70***	0.77	1.00
L+SCS	0.84	0.85	***0.83***	1.00

Table 17: Average Probabilities for Coverage Criteria Pairs

In Table 17, data points are highlighted in bold and italic where the combination of coverage criteria results in a better average probability of detecting each fault than either criterion in isolation. We see such an improvement in over half (27 of 46) of the combinations. This result suggests certain faults may be more highly correlated to different criteria, and thus combining multiple coverage criteria can dramatically reduce fault detection reduction. However, maintaining coverage adequacy with respect to additional criteria in test suite reduction will lead to larger reduced test suites. Indeed, many of the improvements in average probabilities in Table 17 for the GUI subjects involve the addition of the event-interaction criterion, E2, and E2 coverage adequacy in test suite reduction is known to lead to very little

size reduction for these applications and test suites (see Experiment 1, Figures 6 through 8). In test suite reduction, the tradeoff between fault detection and size reduction must be made based on situational engineering judgments.

6.11.5. Summary of Experiment 5

Call-stack-based test suite reduction exhibited several positive attributes in the analyses of Experiment 5, including a high average probability of detecting each fault and a high number of faults always detected after reduction of the test universe. These attributes were more pronounced for the event-driven, GUI subject applications; for the conventional application nanoxml, line-based reduction was the best approach. These results of Experiment 5 do show that certain coverage criteria are more closely related to fault-detecting test cases and therefore may be better suited for use in test suite reduction, thus answering research question Q5.

Chapter 7: Analysis – Test Suite Reduction Metric

Prior work on test suite reduction provides very little guidance for practitioners who must make decisions about what 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 tradeoff in practice. Because of this, there is a need for quantitative metrics that capture the size-versus-fault-detection tradeoff to help guide practitioners needing to make a more holistic choice when applying test suite reduction techniques.

The experiments in Chapter 6 analyzed call-stack-based test suite reduction in terms of size reduction and fault detection reduction independently. In experiments using GUI applications as test subjects, call stack coverage-based reduction resulted in considerably larger reduced suite sizes than various 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, this 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.

In their work on test suite reduction in web applications, Sampath *et al.* [40] propose a "figure of merit" (*fom*) for test suite reduction as:

$$(17)\ fom = redux * cvg * fd$$

Here, *redux* is the percent size reduction, *fd* is the percentage of faults *still detected* after reduction, and *cvg* is the percent 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 Equation (17) 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, this research proposes evaluating test suite reduction relative to the following single-point metric:

$$(18)\ ReductionMetric = (W_{SR} * \%\ Size\ Reduction) + W_{FDR} * (100 - \%\ Fault\ Detection\ Reduction)$$

W_{SR} is defined 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. It is expected that practitioners will choose the weights to capture the relative importance of fault detection and size reduction in a specific industrial scenario.

To demonstrate this new metric, consider three sets of weights defined in Table 18. In Scenario 1, small reduced test suite size is deemed more important than low fault detection reduction. Scenario 2, conversely, considers low fault detection

reduction to be 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.

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

Table 18: Metric Weighting Scenarios

Applying the metric from Equation (4) to the data collected in the experiments from Chapter 6 for the different reduction techniques, subject applications, and weighting scenarios yields the results in Figures 24 through 28.

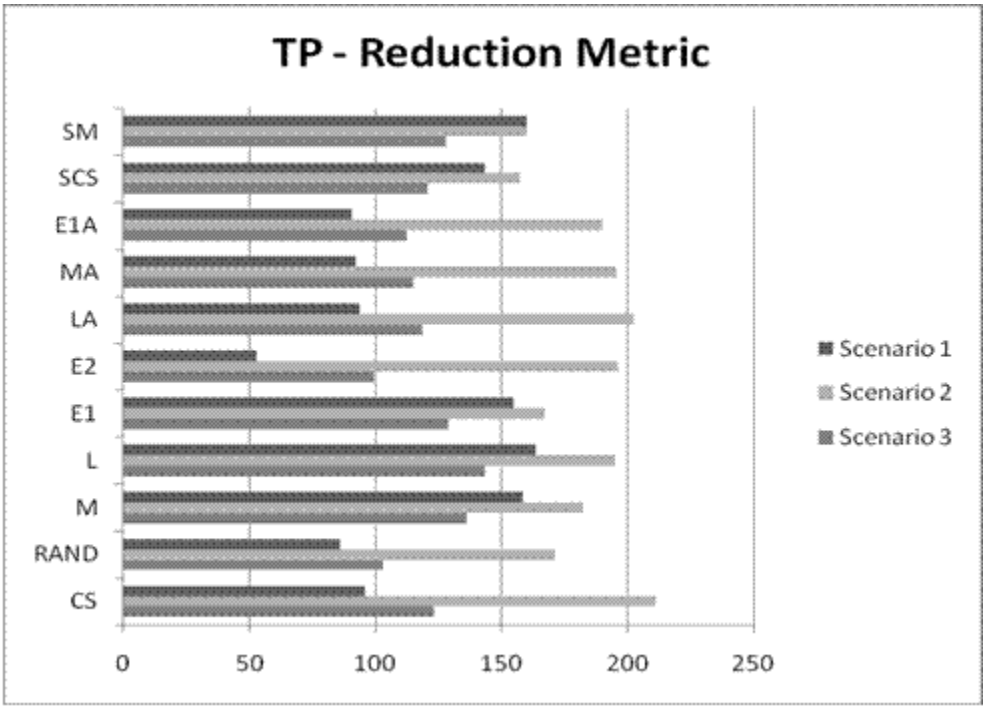

Figure 24: TP Average Test Suite Reduction Metric Over All Suite Sizes

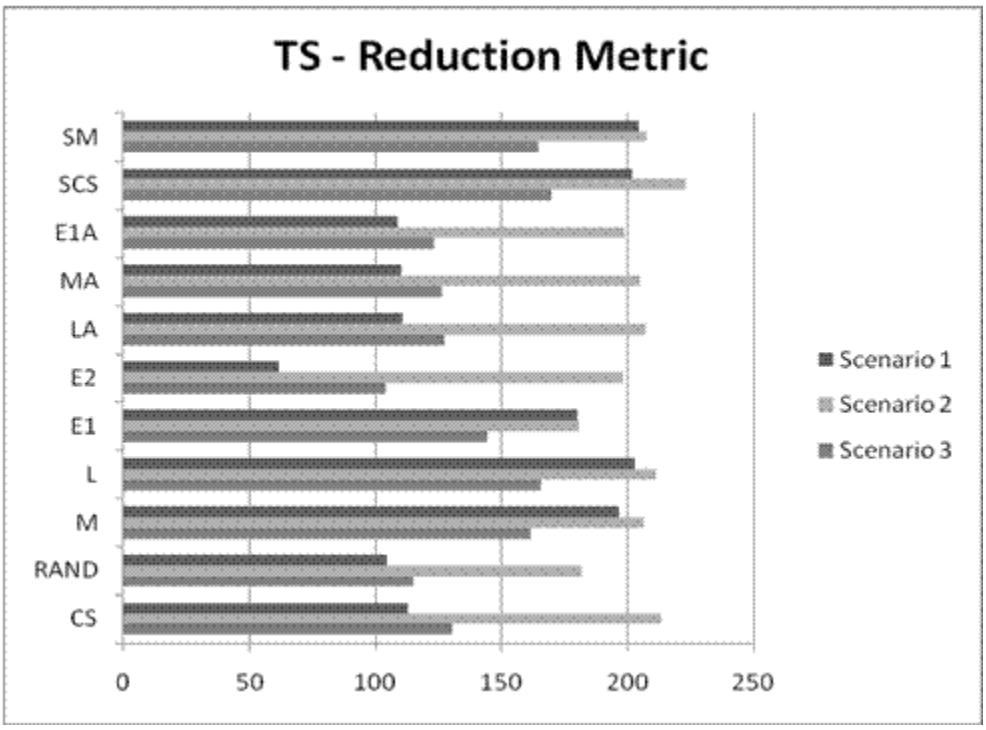

Figure 25: TS Average Test Suite Reduction Metric Over All Suite Sizes

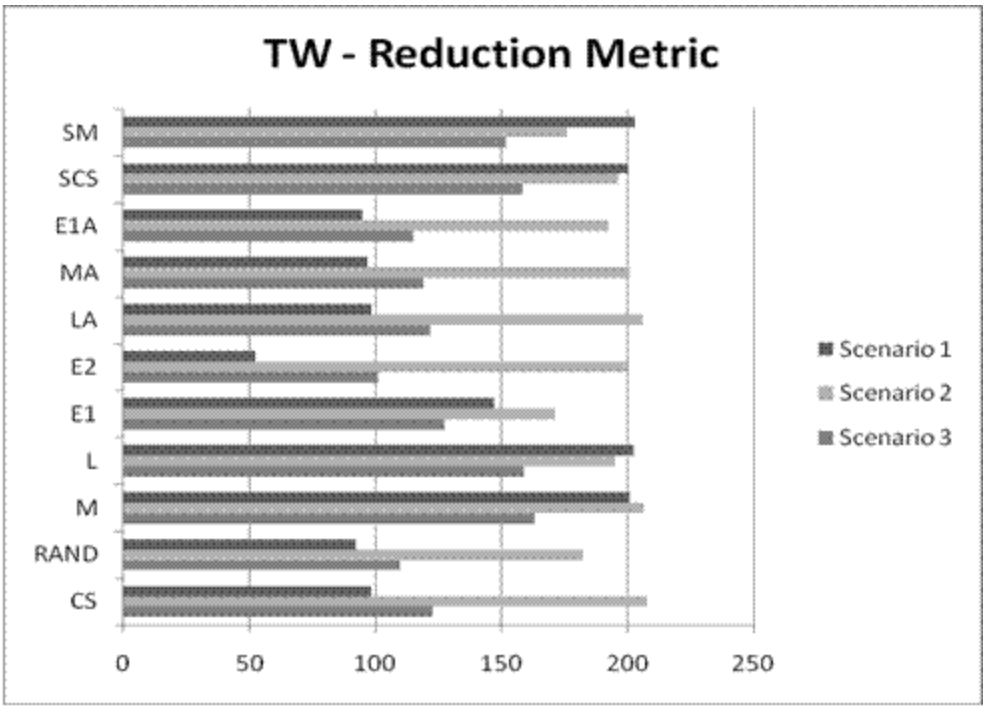

Figure 26: TW Average Test Suite Reduction Metric Over All Suite Sizes

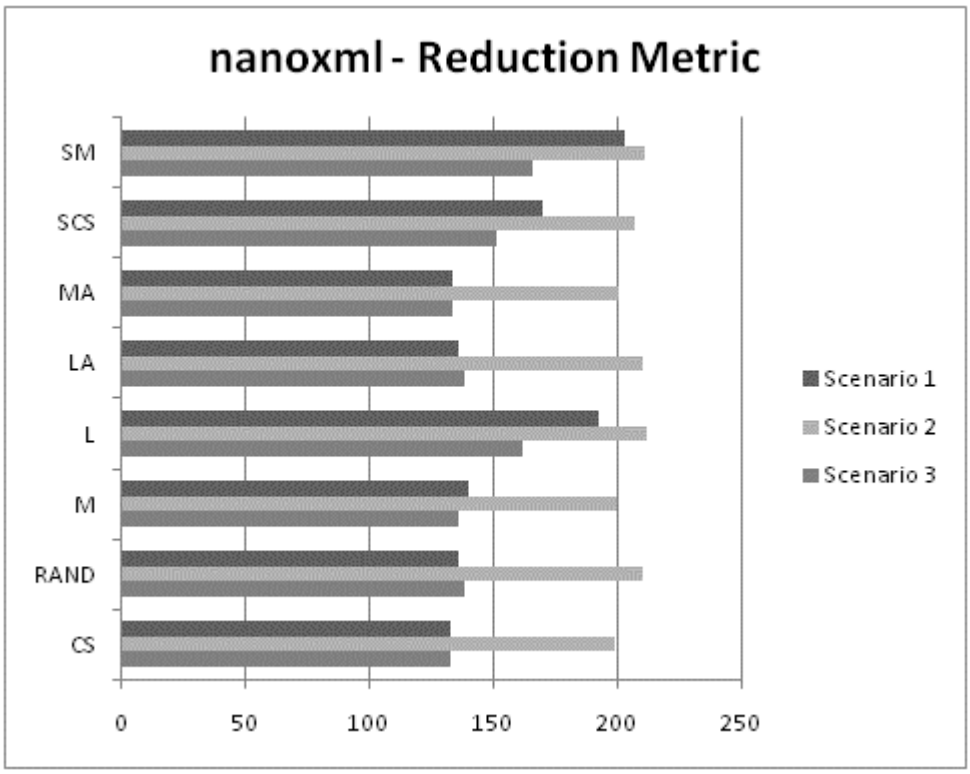

Figure 27: nanoxml Average Test Suite Reduction Metric Over All Suite Sizes

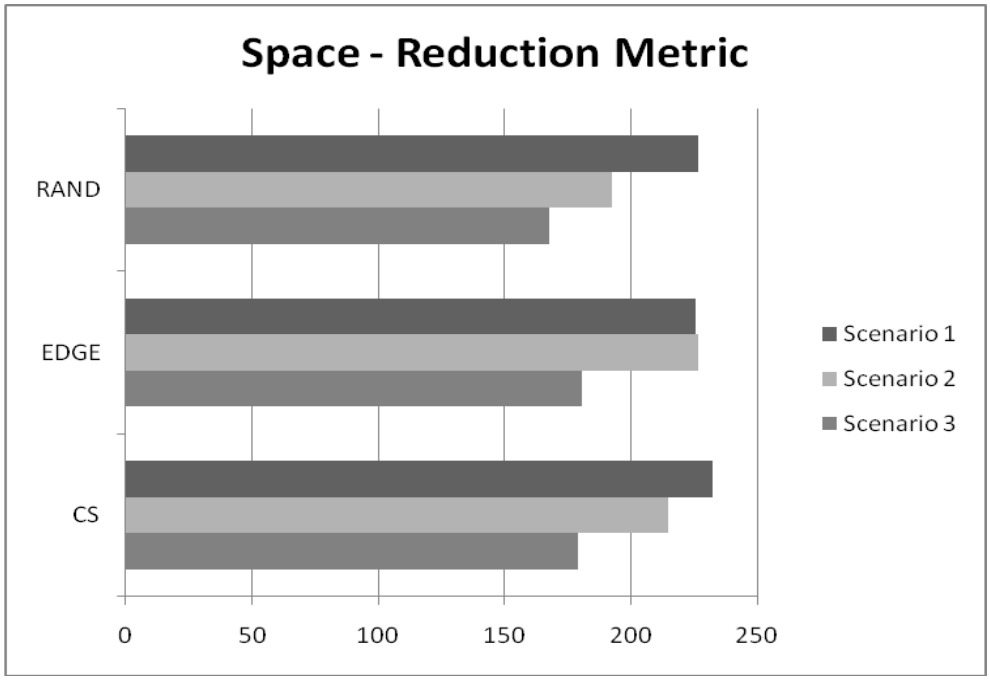

Figure 28: Space Average Test Suite Reduction Metric Over All Suite Sizes

When small suite size is the primary focus of the test suite reduction process (Scenario 1), the metric indicates that the favored techniques for modern GUI applications are based on line coverage (L), method coverage including library methods (M), 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 call stack technique (CS) 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 improved performance of the "additional" techniques MA, LA, and E1A. In nanoxml, the relative performance of techniques is consistent across scenarios with the notable exception that SM performs particularly well when small suite size is the emphasis. And 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 across all scenarios indicate that test suite reduction in general is more effective when applied to TerpSpreadsheet (TS) and TerpWord (TW) than in TerpPaint (TP) or nanoxml. Future work may use this metric in an attempt to identify application construction factors influencing test suite reduction.

Chapter 8: Conclusions and Future Work

This research presented models, metrics, algorithms, techniques, and tools that support a novel approach to test suite reduction based on call stacks. Experiments showed that this approach provides an effective tradeoff between size reduction and fault detection reduction, particularly for modern, event-driven GUI applications. Additionally, collecting and analyzing call stack coverage data was shown to be feasible in non-trivial modern software applications.

It was shown that in most cases, call stack coverage contains valuable information that preserves fault detecting ability of test suites under reduction for modern applications. Indeed, this research has shown that event-driven GUI applications are sufficiently different from traditional applications to benefit from new coverage criteria [29]. This research also found that considering coverage of library methods can improve the fault detection effectiveness of coverage-based reduced test suites.

This work defined a new metric for coverage-based test suite reduction based on the average probability of detecting each fault. This metric was applied to the set of test suite reduction experiments on GUI and conventional subject applications and contrasted the results using several different coverage criteria as well as combinations of criteria. The analysis was extended to count faults detected by a full test suite which must necessarily be detected by any coverage-preserving reduced test suite for the different criteria, and the impact of fault difficulty was also considered. Based on the analysis enabled by the average probability of detecting each fault metric, it was found that certain coverage criteria are more related to fault-detecting test cases than

others, and this behavior varies by application type. In the modern GUI applications used in these experiments, test suite reduction based on call stacks provides the highest probability of detecting each fault in a reduced test suite, method (including libraries), and line coverage perform comparably, and length-1 event sequences are the least effective. This relative ranking was consistent with empirical performance of the various criteria against the traditional percentage fault detection reduction metric. Thus, this research concludes that the average probability of detecting each fault shows promise for identifying coverage criteria that work well for test suite reduction.

Finally, a second metric for test suite reduction based on weighted importance of size reduction versus fault detection reduction was developed and applied to empirical data. A comparison of results between conventional and event-driven GUI applications indicates that the "best" test suite reduction coverage criterion as measured by this metric may differ among other classes and styles of application. It also showed that the choice of coverage criterion for test suite reduction can depend on whether size or fault detection is emphasized.

This work has examined coverage-based test suite reduction for modern software applications from a single (albeit important) perspective, that of the single-user GUI. Additional styles of applications can be classified as "event-driven", including server applications that use concurrent request-response or messaging paradigms, as well as the broader population of distributed and service-oriented computing systems. In these applications, a software component receives a message or method call and optionally changes its state, invokes additional components,

and/or formulates a response. This situation is analogous to how a GUI responds to events. Automatic test case generation techniques that apply to systems of multiple interacting processes have been developed. Yet in the context of such systems, test suite reduction is a less well-studied problem. Therefore, a key direction for future work is to extend and apply the notion of a context-sensitive coverage criterion to test suite reduction and other test case management problems in these systems.

Additional directions for future work in this line of research may include:

- Incorporating new subject applications that represent a wider variety of programming languages and styles, development paradigms, application domains, and sizes.
- Expanding the range of coverage criteria in the comparison, perhaps including techniques less widely used in practice such as advanced dataflow criteria.
- Analyzing characteristics of the faults lost by various reduction techniques to evaluate whether certain types of coding errors are more or less likely to remain undetected in reduced test suites. A further potential consequence of such an evaluation would be quantifying the importance of calling context as represented in call stacks to the test suite reduction problem.
- Applying call stack coverage to other software testing problems, such as test case generation, regression test selection, and test case prioritization.

- Comparing test suite reduction performance using different possible models of call stacks, particularly those that result in lower runtime overhead and coverage data volume. Approaches could include building stacks using the different method representations discussed in Section 3.1.2. Another idea is to define a "similarity metric" for call stacks such that stacks with a certain similarity value could be considered redundant and discarded. Also, certain method calls may always appear together in a call stack so that they could be collapsed into a single stack entry to conserve space and simplify the analysis.
- Incorporating more sophisticated cost models that do not necessarily treat all test cases and all faults equally. A cost-benefit model for defect detection activities has been proposed by Wagner [41], and another model specifically focused on regression testing has been developed by Do and Rothermel [8]. Because of the close relationship between regression testing and test suite reduction, Do and Rothermel's model (which explicitly factors in cost of missing faults and cost of test execution) may be a good candidate to apply to the test suite reduction problem.

Bibliography

[1] Agitar Automated JUnit Generation information on the web at http://www.agitar.com/solutions/products/automated_junit_generation.html, viewed September 2007.

[2] 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.

[3] Beck, K. and Andres, C. *Extreme Programming Explained: Embrace Change (Second Edition)*. Addison-Wesley Professional, 2004.

[4] Bond, M. D. and McKinley, K. S. Probabilistic calling context. OOPSLA'07, October 21–25, 2007, Montreal, Qúebec, Canada.

[5] Cobertura information on the web at http://cobertura.sourceforge.net/, October, 2007.

[6] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. *Proceedings of the 23rd International Conference on Software Engineering*, pages 339-348, 2001.

[7] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. *Proceedings of the International Symposium on Empirical Software Engineering*, August, 2004, pages 60-70.

[8] H. Do and G. Rothermel. An empirical study of regression testing techniques incorporating context and lifecycle factors and improved cost-benefit models. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, November, 2006, Portland, Oregon, USA.

[9] Elbaum, S.; Gable, D. & Rothermel, G. The impact of software evolution on code coverage information. *Proceedings of the IEEE International Conference on Software Maintenance*, 2001, pp. 170--179.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Heimdahl, M. & George, D. Test-suite reduction for model based tests: effects on test quality and implications for testing. Proceedings of the 19th

International Conference on Automated Software Engineering, 2004, pp. 176--185.

[15] J. R. Horgan and S. London. Data flow coverage and the C language. *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, 1991, Victoria, British Columbia, Canada.

[16] G. Hunt and D. Brubacher. Detours: binary interception of Win32 functions. *Proceedings of the 3rd USENIX Windows NT Symposium*, pp. 135-143. Seattle, WA, July 1999.

[17] JavaCCTAgent information on the web at http://sourceforge.net/projects/javacctagent/, April, 2007.

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

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

[20] D. Jeffrey and N. Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Transactions on Software Engineering*, Vol. 33, no. 2, pp. 108-123, February, 2007.

[21] 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.

[22] Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., and Guoliang, Z. Generating test cases from UML activity diagram based on gray-

box method, *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pp. 284-291.

[23] S. McMaster and A. Memon, Call Stack Coverage for GUI Test-Suite Reduction, *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)*, Raleigh, NC, USA, Nov. 6-10 2006.

[24] 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.

[25] S. McMaster and A. Memon. Fault detection probability analysis for coverage-based test suite reduction. *IEEE International Conference on Software Maintenance (ICSM) 2007*, Paris, France, 2007.

[26] 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.

[27] 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.

[28] 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).

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

[30] 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.

[31] J. Offutt, J. Pan, and J. Voas. Procedures for reducing the size of coverage-based test sets. *Proceedings of the Twelfth International Conference on Testing Computer Software*, pages 111--123, June 1995.

[32] Parasoft JTest information on the web at http://www.parasoft.com/jtest, viewed September, 2007.

[33] C.K Prasad, R. Ramchandani, G. Rao, and K. Levesque (June 24, 2004). *Creating a debugging and profiling agent with JVMTI*. Retrieved September 22, 2007, from http://java.sun.com/developer/technicalArticles/Programming/jvmti/.

[34] 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.

[35] 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. *Proceedings of the International Conference on Software Maintenance*, pages 34-43, November 1998.

[36] 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.

[37] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929-948, October, 2001.

[38] 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.

[39] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A. Souter. Applying concept analysis to user-session-based testing of web applications. *IEEE Transactions on Software Engineering*, Vol. 33, No. 10, pgs 643 - 658, October 2007.

[40] S. Sampath, S. Sprenkle, E. Gibson, and L. Pollock. Web application testing with customized test requirements – an experimental comparison study, *Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)*, Raleigh, NC, USA, Nov. 6-10 2006.

[41] S. Wagner. A model and sensitivity analysis of the quality economics of defect-detection techniques. *Proceedings of the ACM International Symposium on Software Testing and Analysis*, July 2006, Portland, Maine, USA.

[42] 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.

[43] 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.

[44] C. Yang and L. L. Pollock. The challenges in automated testing of multithreaded programs. In the *14th International Conference on Testing Computer Software*, pages 157--166, June 1997.