ABSTRACT

Title of dissertation: Defining and Evaluating Test Suite Consolidation for Event Sequence-based Test Cases

Penelope Brooks, Doctor of Philosophy, 2009

Dissertation directed by: Professor Atif M. Memon
Department of Computer Science

This research presents a new **test suite consolidation** technique, called CONTEST, for automated GUI testing. A new probabilistic model of the GUI is developed to allow direct application of CONTEST. Multiple existing test suites are used to populate the model and compute probabilities based on the observed event sequences. These probabilities are used to generate a new test suite that consolidates the original ones.

A new **test suite similarity metric**, called $CONTeSSi(n)$, is introduced which compares multiple event sequence-based test suites using relative event positions. Results of empirical studies showed that CONTEST yields a test suite that achieves better fault detection and code coverage than the original suites, and that the $CONTeSSi(n)$ metric is a better indicator of the similarity between sequence-based test suites than existing metrics.

Defining and Evaluating Test Suite Consolidation for Event Sequence-based Test Cases

by

Penelope A. Brooks

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
2009

Advisory Committee:
Professor Atif M. Memon, Chair/Advisor
Professor Rance Cleaveland
Professor Peter J. Keleher
Professor Adam A. Porter
Professor Brian R. Hunt

Acknowledgments

First, I would like to thank my advisor, Dr. Atif Memon, for his support and guidance in this process. Always willing to listen to my ideas, he was also careful not to let me get too far out of bounds!

Next, my appreciation goes out to a the GUITAR group, past and present. Specifically, I'd like to thank Dr. Qing Xie for recommending Atif and this area of research. Next, thanks to Dr. Xun Yuan for her patience, support, and friendship through countless sessions of my learning the GUITAR suite, finding and installing the subject applications, and running experiments. Bin Gan was very helpful as I learned to run the GUITAR tools. Finally, thanks to Jaymie Strecker who helped me kick off this research area during our class project.

I would be remiss to not mention the US Corporate Research Group at ABB. I am grateful to them for the opportunity to grow as a researcher in this past year.

Finally, and most importantly, I would like to thank my family with all my heart. My husband, who married me just before I started this journey, has endured countless hours of work and discussion of computer science models and methods. I can never repay nor adequately express my gratitude for his undying support and never- ending patience. My parents, always supporting me in whatever I desire, have been nothing short of rocks during this journey. I will never take for granted the love and support these three people have given me in all things.

It is impossible to remember all, and I apologize to those I've inadvertently left out. Thank you.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since the invention of the Graphical User Interface (GUI) [45], it has become the most widely used means of accessing the functionality of a software application. Often, the GUI provides the only method for the user to access the application's functionality. Therefore, it is important to test the application through the GUI, *e.g.*, for system and integration testing, and test the GUI itself for functional correctness. Previous studies have shown that code to implement the GUI can make up as much as 60% of the overall application code [40]. It is important, therefore, to include focused GUI testing as part of the software development cycle. Recognizing this fact, many academic researchers and industry practicioners alike have developed techniques for GUI testing [39, 66, 50, 13, 5].

GUIs are in a class of software called event-driven software (EDS) that take user events[1] as input. GUI-driven applications are typically state-based, meaning that certain events may cause a state change, sometimes enabling or disabling other events. GUI applications are difficult to test due to the large number of events that are legal input to the system, and the enormous number of combinations of events that can be executed as sequences.

[1] An event is a user action that can be performed on a GUI widget, such as clicking on a button or a menu item, or typing in a text field. In the remainder of this document, wherever possible, events will be denoted by their corresponding widget; for example, click_on_Cancel_button will be called Cancel and click_on_menu_item_Save will be called Save.

GUI applications also have constraints regarding the order of events; for example, some events are not allowed in the initial state of the system but are only allowed after a certain sequence of events has been performed. Despite these constraints, the number of test cases that may be executed on the GUI grows exponentially in the length of the test case. Consequently, GUI testing techniques typically sample from the space of all possible sequences of events allowed by the application under test (AUT).

Previous work in GUI testing has involved manually intensive methods of testing. Unit testing tools are the most common manual method used [19]. Unit testing tools require the tester to programatically specify a sequence of program statements that will be executed on the GUI [22]. While this technique has been valuable in detecting faults, due to its labor intensive nature, more automated techniques are desired.

More recent work in GUI testing has focused on automating testing [32, 33, 13]. Two promising techniques are:

1. **Parameterized test case generation:** A fully automatic technique that generates test cases based on the structural model of the GUI, parameterized by the functional units in the application under test (AUT) [66], *e.g.*, *Print function* and *Clipboard function*. The main strength of this technique is that it aids a tester in developing test cases that cover select parts of the GUI. A fundamental limitation of this technique is that it yields short test cases – only length 2 for non-trivial GUIs; obtaining length 3 and above is computationally

intensive. Moreover, it yields a very large number of length 2 test cases – tens of thousands for most applications. Earlier research has shown that these short test cases are effective [39]; however, longer sequences are able to detect new faults missed by short test cases [66].

2. **Usage profiles:** A technique that collects sequences of events from end-users during actual usage of the application using a capture tool. These sequences are then automatically replayed on a new version of the application during regression testing using a replay tool [14, 36, 37, 51]. The main strength of this technique is that it is driven by actual usage of the application, and hence, is able to yield long test cases. However, it requires a fielded system and a user population which agrees to allow monitoring of their application usage. Further, it does not yield test cases containing events and event sequences that users don't execute, which may cause the test cases to miss faults.

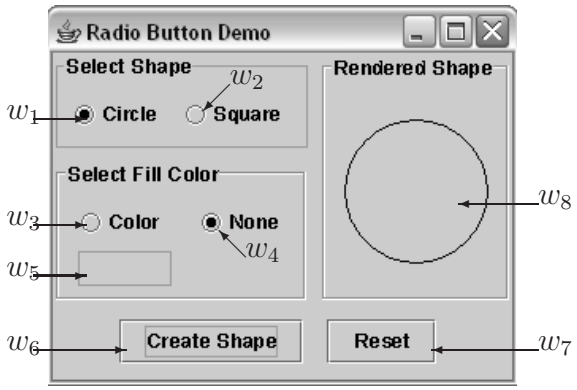


Figure 1.1: A Simple GUI

Consider Figure 1.1, the Radio Button Demo GUI, used to teach programming students how to develop a GUI containing radio buttons. The widgets labeled w_1 through w_7 are those through which users can access the corresponding events

(e_1 through e_7). The *start state* has `Circle` and `None` selected; the text-box corresponding to w_5 is empty; and the `Rendered Shape` area (widget w_8) is empty. Event e_6 creates a shape in the `Rendered Shape` area according to current settings of $w_1 \ldots w_5$; event e_7 resets the entire software to its start state. The other events behave as follows. Event e_1 sets the shape to a circle; if there is already a square in the `Rendered Shape` area, then it is immediately changed to a circle. Event e_2 is similar to e_1, except that it changes the shape to a square. Event e_3 enables the text-box w_5, allowing the user to enter a custom fill color, which is immediately reflected in the shape being displayed (if there is a shape there). Event e_4 reverts back to the default color.

The aforementioned techniques (parameterized test case generation and usage profiles) may be used to test the Radio Button Demo GUI. A test designer using *parameterized test case generation* may divide (with overlaps) the GUI's events into the `Circle`, `Square`, and `Reset` functions; these are used as parameters for the test case generator to yield function-specific test cases. One test case generated by the technique for the `Square` function may be the event sequence $< e_2, e_6 >$, which may be executed in the *initial state* of the AUT – hence it tests event e_6 *after* the event e_2 has executed and modified the AUT's state, thereby testing e_6 in the *context* of e_2's execution. A test case for the `Circle` function may be $< e_1, e_6 >$, which tests event e_6 in the context of e_1's execution. Additionally, the *usage profiles* technique may yield the test case $< e_3, e_5, e_4, e_3, e_6 >$, which tests e_6 in the context of $< e_3, e_5, e_4, e_3 >$. Each of these test cases may reveal faults that require the execution of event e_6 in the specific context established by its preceeding events in

the test case. Hence, all of these techniques are valuable in that they test events and event sequences in specific contexts to detect faults. Additional examples of the test suites generated by these techniques are shown in Figure 1.2. Note that the parameterized technique outputs some length 3 test cases because of the small size of the GUI; usage profiles, on the other hand, yield length 4 and 5 tests. These test suites and the models produced from them will be used as a running example to motivate and explain this research.

Circle	Square	Reset
e_1, e_6	e_2, e_6	e_7, e_1
e_1, e_4	e_2, e_3, e_5	e_7, e_1, e_6
		e_7, e_2, e_6
		e_7, e_2, e_3

(a) Parameterized Technique

Usage Profiles
e_1, e_3, e_5, e_6
e_2, e_3, e_5, e_6
e_2, e_4, e_7, e_2, e_6
e_3, e_5, e_4, e_3, e_6
e_7, e_2, e_3, e_5, e_6

(b) Usage Profiles

Figure 1.2: Example test cases.

There are several other semi-automatic techniques to generate GUI test cases based on a model of the AUT; the model is typically created by hand – a resource intensive activity. Examples include techniques based on exploring the state of the AUT [39, 66], AI planning [38], genetic algorithms [25], and finite-state machines [54] and their extensions [50] to generate test cases. The state-based approach yields sequences that explore new, untested states, thereby revealing faults in these new states [66, 39, 6]. AI planning yields test cases that cover specific use cases [38]. Genetic algorithms yield test cases to mimic specific user populations [25]. Because of the resource intensive nature of these techniques, they yield a small number of long tests.

In practice, a GUI test designer may use a mix of the above techniques to

obtain several test suites. The test designer is faced with two significant challenges:

- *Overlaps in test suites:* As one can imagine, many of these techniques often overlap in what they test. A test designer who uses two or more GUI testing techniques may waste valuable resources testing and retesting the same parts of the AUT. Ideally, the test designer would like to consolidate all the test suites and obtain one suite that minimizes overlaps.

  An existing solution is to merge all the test suites produced by the individual techniques into one suite and use test suite reduction [16] (or test suite minimization [48, 59]) to yield a single, reduced test suite. In these techniques, test cases are selected based on measurable criteria inherent in running the test suites, such as statement [65], MC/DC code [21], and call-stack coverage [34].

  This is not a viable solution for GUI testing because of several reasons, all rooted in the event driven nature of GUIs. The most important reason is that events need to be executed in various contexts because fault detection is often context dependent [62, 34]. All of the GUI test case generation techniques summarized earlier force certain events and event sequences to be executed in specific contexts. Test reduction that ignores context-specific issues will compromise fault detection.

  Next, some of these issues are informally explored via a running example. All of the original suites, given in Figure 1.2, were merged to obtain a single suite shown in Column 1 of Table 1.1. Next, the popular HGS algorithm [17] was

used to reduce this merged suite.[2] One requirement for reducing the suite was event-pair coverage, which retains test cases that cover unique pairs of events. The suite was also reduced using several code-based criteria for the reduction. Columns 2-5 of Table 1.1 show the reduced suites. Event-pair coverage is the only criterion that considers context, albeit of a single event, during reduction. However, each of these reduced suites eliminate tests that may potentially detect faults. More importantly, recall that each test case generation technique individually advocates the execution of certain sequences of events in specific contexts. For example, the subsequence $< e_7, e_2 >$ appears four times in the original suites; $< e_2, e_3 >$ appears four times; $< e_1, e_6 >$ appears twice; they all appear in multiple contexts. Reduction does not consider preserving the importance of these frequencies and/or contexts. This is precisely the reason that reducing a GUI test suite based on code coverage has been shown to reduce the effectiveness of the suite [34]. Several researchers have reported that test suite reduction methods have decreased fault detection while maintaining the same level of code coverage [21, 48, 18, 65].

- *Large number of short tests and few long tests:* The sheer size of the individual suites presents practical problems for test execution. Because each test case requires significant overhead in terms of *setup* and *teardown*, having a large number of short tests is inefficient. Ideally, the test designer would like to obtain longer sequences that combine the strengths of individual short-sequence

[2]The HGS algorithm is a greedy test suite reduction algorithm that iteratively examines test cases and saves those which satisfy the most as-yet-unsatisfied requirements. A requirement is removed from the list when one test cases satisfies it. Full details of this algorithm can be found in [17].

Original Merged Tests	Event Pair Coverage	Line Coverage	Method Coverage	Branch Coverage
e_1, e_6	e_1, e_4	e_7, e_1, e_6	e_1, e_3, e_5, e_6	e_1, e_4
e_2, e_6	e_1, e_3, e_5, e_6	e_7, e_2, e_6	e_7, e_2, e_3	e_7, e_1, e_6
e_7, e_1	e_2, e_4	e_2, e_3, e_5, e_6		e_7, e_2, e_6
e_1, e_4	e_3, e_5, e_4, e_3, e_6	e_3, e_5, e_4, e_3, e_6		e_2, e_3, e_5, e_6
e_2, e_3, e_5	e_7, e_2, e_3	e_7, e_2, e_3, e_5, e_6		
e_7, e_1, e_6	e_2, e_3, e_5			
e_7, e_2, e_6	e_2, e_6			
e_2, e_4 e_7, e_2, e_6	e_7, e_1, e_6			
e_7, e_2, e_3				
e_1, e_3, e_5, e_6				
e_2, e_3, e_5, e_6				
e_3, e_5, e_4, e_3, e_6				
e_7, e_2, e_3, e_5, e_6				

Table 1.1: Example test cases yielded from several reduction techniques

suites.

One solution is to *concatenate* multiple event sequences together. For example, $< e_7, e_1 >$ from the `Reset` suite may be joined with $< e_3, e_6 >$ from the `Circle` suite to obtain a single test case $< e_7, e_1, e_3, e_6 >$; the original tests may be eliminated. Although this join operation reduces the number of test cases, it has several problems. First, certain event sequences may be disallowed by the GUI – simply joining two sequences might yield an unexecutable sequence. Second, the elimination of the original tests may cause the new suite to miss faults that might have been detected had $< e_3, e_6 >$ been executed in the GUI's initial state. Third, the joined sequence tests new interactions, *e.g.*, $< e_7, e_1, e_3 >$, $< e_1, e_3, e_6 >$ that were not advocated by any of the original suites. Although this may be desirable in certain situations, it would lead to a large number of resulting test cases. Resources may be better spent on obtaining long sequences composed of short sequences that are advocated by

the original suites. For example, the subsequence $< e_7, e_2 >$ appears thrice in the original suites; $< e_2, e_3 >$ appears five times; $< e_3, e_6 >$ appears twice. A long sequence *e.g.*, $< e_7, e_2, e_3, e_6 >$ composed of these subsequences would at least ensure that these individual sequences are covered. Other sequences would also be needed to ensure that all the sequences frequently advocated by the original suites are covered.

This research focuses on addressing the challenges and goals given above, and proving the following thesis statement: **A method of combining and consolidating sequence-based test suites preserves the context observed in the existing test suites and maintains their fault detection effectiveness.**

More specifically, the thesis statement is proved by developing a new method of *consolidating* existing test suites, based on the characteristics of the existing suites and the structural relationships between events. From the original suites, the probabilistic relationships between individual events, *i.e.*, the conditional probability that one event is executed after another event, are computed. Further, the probabilistic relationships between sequences of events, *i.e.*, the conditional probability that one event is executed after a sequence of events, is also computed. These relationships are captured in a new model of the GUI, which is then used to generate test cases.

A new algorithm called CONsolidate TEST suites (CONTEST) is developed which takes existing test suites as input and generates a single consolidated test suite based on the computed conditional probabilities. CONTEST is evaluated by conducting an empirical study on four open source GUI applications. For each ap-

plication, two types of test suites are used. First, a number of suites are generated using the parameterized test-case generator; the number varies by application, ranging from 8 to 18 suites. Second, one test suite per application is developed from usage profiles.

Beyond measuring the benefits of CONTEST via traditional metrics based on fault detection and size, it is important to further evaluate the CONTEST algorithm by performing a suite-to-suite comparison of the suites consolidated by CONTEST to the existing event-based suites on the basis of the events that are executed in each suite. To perform this comparison, a new metric was developed as part of this research which considers the characteristics of event-driven systems, including the context and state-based execution of the running application. This metric is based on a popular Information Retrieval (IR) measure, *cosine similarity*. This metric, called $CONTeSSi(n)$ (CONtext Test Suite Similarity), explicitly considers the context of n preceding events to measure the similarity between suites.

As a precursor to the development of CONTEST and $CONTeSSi$, the effectiveness of crash testing for GUI systems was researched to improve understanding of how to test them. The term *crash testing* refers to testing which uses a crash in the running application as the *test oracle* (a mechanism that determines whether a test case passed or failed). For the purposes of this research, a crash is defined as an uncaught exception thrown during test case execution. Test and field defects discovered on industrial systems were examined to ascertain the effectiveness of crash testing for GUIs.

1.1 Summary of Studies

In total, three separate empirical studies were performed to prove the thesis statement. The summary of these studies is outlined here.

1.1.1 Crash Testing Effectiveness

First, a study was designed and conducted to learn more about the effectiveness of using crash testing to test GUIs since the other two studies were designed to use crash testing. In particular, this study was conducted to characterize GUI systems based on artifacts from testing and development, including source code measures, change metrics, and test suite characterization. A thorough analysis of the defects discovered in three fielded, industrial applications, was performed by focusing on defects detected in or through the GUI front-end. A modified version of Beizer's taxonomy [3, 9], discussed in Section 3.1.1, was used to classify the defects. This study showed that crash testing is an effective way to find defects in the GUI itself as well as defects in the underlying application. It also laid the foundation for the second and third studies.

1.1.2 Consolidating Test Suites

Next, a study was designed to investigate the main thrust of the thesis statement: test suite consolidation. This study examined the effectiveness of the CONTEST algorithm. The CONTEST algorithm was implemented and used to generate test suites for four subject applications. The existing test suites were then compared

to those generated by CONTEST on the dimensions of size, fault detection and code coverage. The study also investigated tuning the model creation and test case generation of CONTEST. As a parameter to the model, the event context upon which conditional probabilities were based (`history`) was varied from 1 event to 5 events. This study showed that the probabilistic model-based approach used in CONTEST to consolidate test suites is effective at preserving the context of the existing suites and at least maintaining their fault detection effectiveness.

1.1.3 Measuring Test Suite Similarity

Finally, a study was designed to compare the consolidated suites to existing suites. This study extended the second study, by utilizing the same test suites, to evaluate $CONTeSSi(n)$. The $CONTeSSi(n)$ metric allows a test designer to evaluate the similarity between test suites by measuring the number of differences between the suites in terms of relative event positions. In this study, the value given by the metric was compared to existing measures of similarity such as code coverage. The results of this study demonstrated that a context-based metric provides a valuable method to compare sequence-based test suites.

1.2 Contributions

The work presented in this document makes the following primary research contributions:

1. introduction of the concept "consolidation of test suites" for event sequence-

based test cases,

2. a new probabilistic representation of a GUI application that combines its static window/widget structure with legal event sequences,

3. a method to populate the model with event sequences from multiple sources,

4. an algorithm to generate test cases based on the model,

5. a metric which allows the comparison of sequence-based test suites using explicit context,

6. an empirical study of three large, deployed, industrial applications comparing 1,215 defects in these applications, classified using a modification of Beizer's defect taxonomy and yielding a characterization of GUI systems,

7. an empirical study comparing the cost, fault detection ability, and code coverage of a test suite generated by the CONTEST technique to test suites generated by other methods on four open source applications,

8. an empirical study in tuning model creation by varying the number of events upon which probabilities are conditioned, and

9. an empirical study demonstrating the effectiveness of the first context-based similarity metric for sequence-based test cases.

1.3 Intellectual Merit and Broad Impacts

The insights into using crash testing on GUI software can be used by software practitioners and testers to better plan for testing their applications. While some researchers have questioned the effectiveness of crash testing in the past, this work provides evidence using industrial systems that crash testing is effective for GUI testing, thereby paving the way for other software researchers to use this technique.

This is the first time the idea of test suite consolidation has been introduced. The model and algorithm that make up CONTEST are the first of their kind and can be extended to domains other than GUI testing. Researchers may apply the CONTEST approach to test suites for conventional software, web applications, and embedded applications.

Likewise, this is the first time a metric has been developed to compare whole sequence-based test suites. With very few modifications, researchers may also extend the metric presented here to a weighted metric where certain criteria are more heavily counted in the comparison.

Although this research focuses on testing EDS systems and specifically GUIs, this work has a broader impact, as event driven software is used by many devices, from cell phones to cars to home appliances. Software testers are always looking for better ways to verify and validate their software and this work provides them with a new approach to testing EDS.

1.4 Published Works

Each of the concepts presented in this document has been published. The idea of characterizing GUIs and determining the effectiveness of crash testing for GUIs was published at the IEEE International Conference on Software Testing, Verification and Validation in 2009 [5]. The first version of the CONTEST algorithm was published in 2007 at the IEEE/ACM International Conference on Automated Software Engineering [6]. The $CONTeSSi(n)$ metric has been accepted for publication at the IEEE International Conference on Software Maintenance [7].

1.5 Structure of this Document

Next, Chapter 2 provides background on GUI testing and fundamentals, followed by related works in areas pertinent to this research. Chapter 3 presents a study on the effectiveness of GUI crash testing. Chapter 4 presents the concept of test suite consolidation and an empirical study showing its effectiveness. Following that, Chapter 5 introduces a metric allowing the comparison of test suites and an empirical study showing the effectiveness of the metric. Finally, Chapter 6 describes the conclusions and future research directions of this work.

Chapter 2

Background and Related Work

This chapter presents relevant background regarding GUI fundamentals and related work in the areas of software testing, defect classification, and similarity metrics.

2.1 Background

GUI fundamentals are best explained with a visual example. Figure 2.1 shows the Radio Button Demo GUI, first presented in Chapter 1 and shown here for ease of the reader.

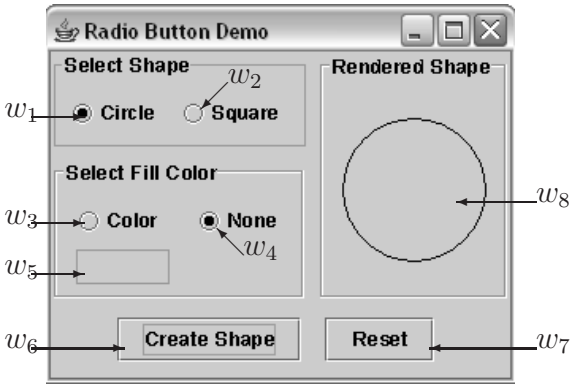


Figure 2.1: A Simple GUI

A GUI is modeled as a set of *widgets* $W = \{w_1, w_2, ..., w_l\}$ (e.g., buttons, panels, text fields) that constitute the GUI, a set of *properties* $P = \{p_1, p_2..., p_m\}$ (e.g., background color and shape) for each of these widgets, and a set of *values* $V = \{v_1, v_2..., v_n\}$ (e.g., red and square) associated with the properties. Each GUI

will contain certain types of widgets with associated properties. As described in Chapter 1, the GUI in Figure 2.1 has 7 widgets with associated events. The state of a GUI can be specified at any time during its execution as a set S of triples (w_i, p_j, v_k), where $w_i \in W$, $p_j \in P$, and $v_k \in V$. A description of the complete state of the GUI contains information about the types of all the widgets currently in effect in the GUI, as well as all of the properties and their values for each of those widgets.

Each particular GUI has a distinguished set of states called the valid initial state set, S_I; the GUI may be in any state $S_i \in S_I$ when it is first invoked. Some GUI events are not available until another event or event sequence is executed. In the GUI shown in Figure 2.1, the text box (w_5) under the radio button `Color` (w_3) is not active until after the radio button is selected. Therefore, users may not execute event e_5 in the initial state of the GUI.

The state of a GUI is not static; events performed on the GUI change its state. These states are called the reachable states of the GUI. The events $E = \{e_1, e_2, \ldots, e_n\}$ associated with a GUI are functions from one state to another state of the GUI. These events may be strung together into sequences, as permitted by the GUI structure, called *legal event sequences*. A legal event sequence of a GUI is $e_i; e_{i+1}; e_{i+2}; \ldots; e_{i+n}$ where e_{i+1} can be performed immediately after e_i.

Finally, a GUI test case can be defined in terms of the preceding constructs: a GUI test case T is a pair $< S_0, e_L >$, consisting of a state $S_0 \in S_I$, called the initial state for T, and a legal event sequence e_L.

Figure 2.2 shows an initial state of the GUI (S_0) in the upper left corner. If

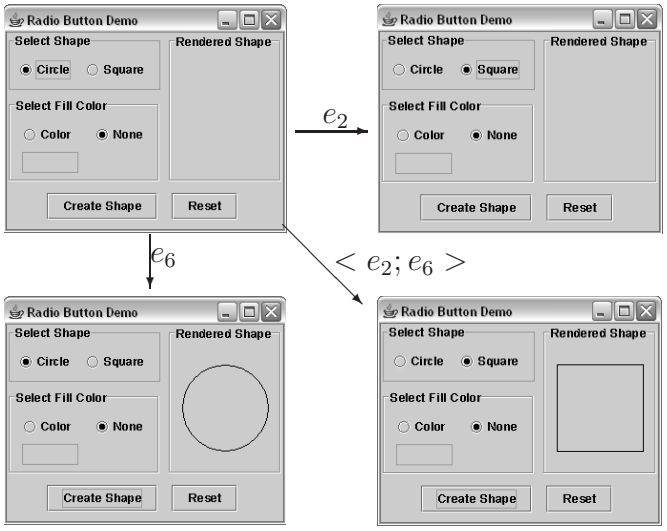


Figure 2.2: Execution of Events e_2 and e_6

event e_2 (click on `Square` radio button) is executed on this GUI, the state of the GUI changes to that seen in the upper right corner of Figure 2.2. In this state, `Square` is set and `Circle` is reset. If event e_6 (click on `Create Shape` button) is executed from S_0, the GUI changes to the state shown in the lower left corner of Figure 2.2, and a circle appears in the Rendered Shape section of the GUI. Furthermore, if the aforementioned events e_2 and e_6 are executed in the sequence $< e_2; e_6 >$, the state of the GUI changes to that seen in the bottom right corner of Figure 2.2; a square is rendered. This small example supports the intuition that certain events affect the state of the GUI differently in sequence than alone.

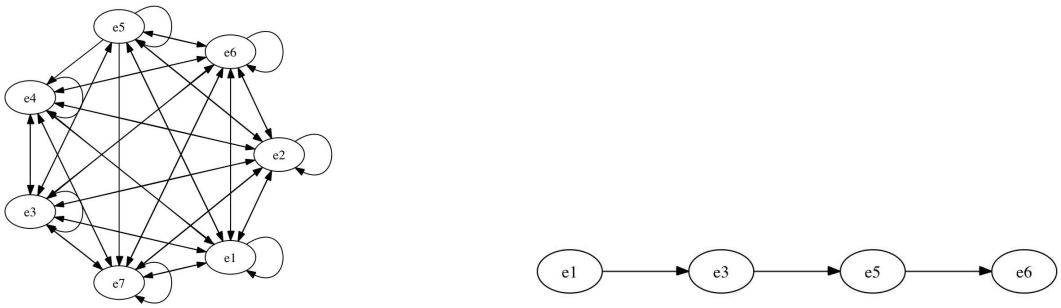
2.1.1 Modeling the GUI

It is desirable to develop a single multi-purpose model that can be used to represent all possible GUIs. Due to the variety of applications with a GUI front-end, however, it is challenging to come up with such a model. Therefore, one sub-class of GUIs is modeled in this research: those which take input from a single user, have

a fixed number of events, and are deterministic, *i.e.*, the outcome of each event is completely predictable. This class of GUIs can be represented by an *Event-Flow Graph* (EFG) and standard graph walking techniques can be used to reason about the model and generate test cases [39].

An EFG is a specific model of the GUI for a particular application, representing all possible sequences of events that a user can execute on that GUI. Nodes in the EFG represent events, and directed edges represent the *event-flow* relationship between two events. That is, an edge in the graph from event e_1 to e_2 indicates that event e_2 may be invoked *immediately after* event e_1. The predicate $follows(e_2, e_1)$ represents this relationship and denotes that e_2 *follows* e_1. EFGs are potentially cyclic, since events can typically be executed more than once during a session with an application. For instance, revisiting the example from Chapter 1, the simple GUI in Figure 2.1 can be represented as the EFG in Figure 2.3(a), which shows the *follows* relationship of the events in the GUI. Because the GUI is quite flexible, the user can perform almost all events immediately after almost all other events, which accounts for the large number of edges in the EFG. Note that the graph is not fully connected; e_5 cannot be executed after e_4, nor can it be executed after e_7.

For the purpose of the work presented in this document, an EFG can be *tailored* to represent only a few sequences of events, thereby creating a sub-graph of the EFG. For example, one sequence of events may select the shape of square, select a fill color, type a fill color in the text box, and create the shape. Corresponding to the widgets in the GUI, this sequence of events is $< e_1, e_3, e_5, e_6 >$. A tailored EFG model (shown in Figure 2.3(b)) of this GUI would represent each of these events as



(a) EFG for the Radio Button GUI (b) Tailored EFG based on example event sequence

Figure 2.3: Radio Button GUI used to demonstrate the static GUI model

nodes while capturing the flow of events between the nodes.

The concepts presented in this section will be used in the rest of this document.

2.2 Related Work in Software Testing

The research presented in this document falls under the broad umbrella of software testing. Based on a search of the literature in this area, the concepts of test suite consolidation and test suite similarity metrics are new – no one else has attempted these before. Therefore, in this section, the broadly related areas of GUI testing, probabilistic testing, state machine model testing, and test suite reduction will be discussed due to the following reasons. First, this research focuses on testing EDS systems, primarily those with a GUI front-end; preceding techniques used to test GUIs are discussed (Section 2.2.1). A key component of this research is the state-based probabilistic Markov model that is created and used to consolidate test suites; therefore, previous research in probabilistic testing and state machine model testing are discussed next (Section 2.2.2). Finally, test suite reduction is broadly related to this research due to its goal of shrinking a suite based on a set of criteria,

and is discussed (Section 2.2.3).

2.2.1 GUI Testing

There are a variety of popular techniques used for testing GUIs, including test harnesses, capture/replay tools, and model-based methods. The first approach involves using test harnesses to test the application. Test harnesses invoke methods in the underlying business logic of a application, as if executed by the GUI, without actually using the GUI [33]. While test harnesses effectively isolate the behavior of the GUI, they do not test the interface code itself and, therefore, are not relevant to this research.

The second approach employs manual tools to execute events directly on the GUI. Popular examples include GUI extensions of unit testing tools and capture/replay tools. JUnit, an open-source, unit testing framework used to test Java code, has been extended into JFCUnit and Jemmy Module for use in GUI testing [22]. Several capture/replay tools have also been developed including Winrunner[1], Abbot[2], and Rational Robot[3] [22]. These tools mimic actual usage of a GUI by recording interactions with the GUI in *capture* mode and replaying the interactions in *replay* mode. A common approach of capture/replay tools is to store mouse coordinates, causing test cases to be fragile and dependent on the GUI layout, and rendering many test cases ineffective for regression testing since the GUI layout may change between versions. Some tools avoid this problem by capturing GUI widgets rather

[1]Mercury Winrunner, http://www.mercuryinteractive.com/products/winrunner
[2]Abbot JavaGUITest Framework, http://abbot.sourceforge.net
[3]Rational Robot, http://www.rational.com.ar/tools/robot.html

than mouse coordinates. The trade-off to this approach, however, is that a significant amount of manual effort is required for the test cases to be effective, including developing test scripts and manually detecting failures. Tools that use scripts also suffer from the problem that any modifications to the GUI require changes to the scripts as well [13, 54]. Testers who employ these tools typically come up with a small number of tests to utilize [35].

The third approach uses a graph model of the GUI to generate test cases. Previous research has shown the usefulness of graph models to represent the GUI, and test cases were developed by exploring paths through the model [12, 42, 39, 66]. Both Xie *et al.*, and Yuan *et al.*, have researched methods to shrink the state space of the GUI and generate test cases based on a model of the GUI. Xie *et al.*, developed a graph model of a GUI and used it to generate test cases [39]. Graph models can be walked, either randomly or deterministically, but cannot be traversed exhaustively because of the large number of possible traversals. Therefore, Xie *et al.*, traversed all paths of the graph up to a specified length, generating test cases that covered the whole application.

Yuan *et al.*, developed a model-based testing approach to automatically generate test cases from a subset of the events in the GUI [66]. Edges in the graph model are annotated based on GUI event interactions learned from a seed suite of test cases, effectively partitioning the graph of the GUI. Run-time feedback provides input to the test case generation procedure based on the annotated graph. The resulting test cases, longer than those generated previously, are generated exhaustively for each partition, allowing the new test cases to model complex GUI behaviors.

Several researchers developed tools that the tester must interact with to generate a graph model of the GUI. Dalal *et al.*, created a tool to capture the functional model of the data, guided by the structure of the GUI [12]. The tool represents the functional model as a graph and generates test cases based on valid inputs to the model. Ostrand *et al.*, have developed a test development environment (TDE) for GUI-based applications [42]. The output of the TDE is a *top-level* graph model of the GUI, in which each node represents a window and each edge represents a user action. A user may drill-down into the *component representation* of the GUI, which models each window in more detail. To generate test cases, a tester can interact with the GUI model, or the TDE can capture users' interactions with the GUI and replay them. Both of these methods of generating test cases from the graph model are manually intensive efforts for the tester.

2.2.2 Testing using Probabilistic Models

State machines, and their extensions to encode behavioral models, usage models, and operational models, are well-known ways to represent complex software. State machine models, in which nodes represent program states (or sets of related states) and edges represent transitions between states, are the most common type of models which encode the behavior of a program. Each test case is generated by traversing the model, representing the path traversed [10]. Walton *et al.*, described an eight-step methodology for creating a usage model. In developing the usage model, the software specification is used to identify and model expected usage,

classes of users, and environment parameters [53].

Markov chain models allow the model to be reasoned about using the general Markov property, *i.e.*, the next state is independent of all past states given the present state. This property helps to control the state explosion problem. Markov chains can be implemented and manipulated as a table, which makes implementation easier, where each row and column represents states; transition probabilities are filled in the table cells for valid transitions between states. Transition probabilities may be assigned in three ways: uninformed, informed, and intended [55]. The *uninformed* approach uses a uniform distribution across possible transitions from any given state, while the *informed* approach assigns probabilities based on observed user data. Probabilities assigned using the *intended* approach compute probabilities based on an average user in the field. For any given application, several models can be developed to represent different classes of users, and probabilities are assigned accordingly.

Whittaker and Thomason developed a Markov model version of the operational model, in which the probability of visiting a node depends only on the previously visited node, and generated test cases stochastically [56]. Özekici and Soyer extended this idea further using a Bayesian framework, in which model parameters can be learned and updated during testing [44]. Woit extended the Markov model to better handle conditional probabilities [58, 57]. Woit's approach more accurately models software for which event probabilities depend on a longer history of events. To define model parameters, Woit's method requires the space of events to be manually partitioned into a manageable number of subsets. Probabilities for each subset

are computed by observing users in a testing environment or by estimating based on discussions with users. The probabilities are manually entered into the model. Test cases are generated by choosing a path through the model stochastically using the probability distribution observed or estimated for actual usage.

2.2.3 Test Suite Reduction

Although the research presented in this document is not focused on test suite reduction, there is a relationship between this research and test suite reduction. The primary similarity is that both test suite consolidation and test suite reduction have the goal of changing a test suite (or multiple suites) while still maintaining some characteristics of that suite, usually fault detection effectiveness and code coverage. In the literature, there is only one paper that presented test suite reduction in GUIs [34].

Most of the test suite reduction efforts are focused on non-GUI software. Research in test suite reduction has focused on determining the criteria upon which test cases will be saved or discarded. Heimdahl and George [18] focused their research on test case generation and testing formal specifications. They could drastically reduce test suites while maintaining code coverage; however, fault detection was adversely affected. Other researchers had a similar experience. Two separate efforts, reported by Rothermel *et al.*, and Jones and Harrold, found that while maintaining code coverage in reducing the test suites, fault detection was sacrificed [21, 48]. Conversely, Wong *et al.*, succeeded in minimizing the test suites without a reduction in fault

detection effectiveness by using coverage as the reduction criteria which must be upheld [59].

Another body of research compared several techniques of test suite reduction in an attempt to find a balance between test suite reduction and fault detection effectiveness [65]. Several common methods of test suite reduction were studied, including retaining test cases that exercise more dynamic basic blocks, test cases that execute the most statements or blocks of code, test cases that provide the best MC/DC coverage, and random selection of passing and failing test cases. Hao *et al.*, found the similarity of test cases in a suite also impacts fault detection effectiveness; test case redundancy results in a loss of fault detection effectiveness [15].

Finally, McMaster *et al.*, researched test suite reduction specifically for GUI applications through the use of the call stack [34]. Traditional methods for test suite reduction have not been effective, primarily due to multilanguage implementations of GUIs, the nature of event handlers, reflection, and multithreading. By using the call stack as a coverage criterion, McMaster *et al.*, found a measurable reduction in the size of the suites and a very small difference in fault detection.

2.3 Related Work in Defect Classification

Chapter 3 presents research on the effectiveness of crash testing for GUIs, and relies heavily on a tester's ability to classify defects. A literature search in this area did not produce any research papers specifically characterizing GUI applications, industrial or open source, through studying their defect profiles and source code

characteristics. There are, however, several other areas of research related to the classification of GUI defects, including research on defect classification schemes (Section 2.3.1) and case studies of defect classification (Section 2.3.2). Previous work in these areas helped shape the study executed as part of this research.

2.3.1 Defect Classification Schemes

Many taxonomies exist for classifying software defects, including those described in [3], [20],and [43]. One of the best known taxonomies was developed by Boris Beizer [3] and has eight categories of defects: *requirements, implemented functionality, structural, data, implementation, integration, system and software architecture, and setup* and *test*. Each defect category is further refined into three levels of subcategories, allowing defect classification to be very precise.

While these taxonomies are designed to be generally applicable, other taxonomies are more specialized. Binder [4] describes one that has been specifically tailored for object-oriented programs, whereas Vijayaraghavan and Kaner [52] focus on eCommerce applications. Knuth describes a more course-grained schema based on the errors found in the TEX typesetting system [28]. Another taxonomy was developed for faults in user requirements documents [49], and still others discuss hierarchies of faults present in Boolean specifications [23, 29].

The IEEE Standard Classification for Software Anomalies [20] presents a process for handling and resolving software defects as well as a taxonomy for classifying them. This classifies both the source of the defect (*i.e.*, Specifications, Code, etc.)

and the type of defect (*e.g.*, Logic Problem). However, it is not as detailed as the other taxonomies mentioned above.

In contrast to the taxonomies described above, Orthogonal Defect Classification (ODC) [9] allows practitioners to categorize defects according to their type and tie each one to a phase in the software development cycle where the defect could have been caught, generally with less impact on the software product. ODC has eight defect types: *function, interface, checking, assignment, timing/serialization, build/package/merge, documentation* and *algorithm*. These are associated with nine stages of software development: *design, low-level design, code, high-level design inspection, low-level design inspection, code inspection, unit test, functional test*, and *system test*. ODC uses fewer, more general defect categories than other schemes and is primarily focused on process improvement, rather than statistical defect modeling.

2.3.2 Defect Classification Studies

Several researchers have conducted defect classification studies [8, 40]. Three case studies were presented by IBM in 2002, which illustrated the success of characterizing defects in improving software testing strategies for large, deployed projects [8]. As part of the case studies, periodic assessments became part of the software process, which allowed the organizations to better see the cause of defects, thereby allowing them to change their processes early and prevent late-phase defects.

Other research has involved manual examination of code post-development. Using code inspections and the associated change history on software developed

for an undergraduate course in high performance computing, another study relied on manual efforts to document defects, classify the defects based on a six-category classification scheme, and then develop hypotheses on how each defect type could be avoided [40].

2.4 Related Work in Similarity Metrics

A search of the literature reveals that no test suite comparison metric has been developed for event-based test suites. Research in the information retrieval (IR) and natural language processing (NLP) communities, however, is closely related to the concept of comparing test suites. Research in IR and NLP focuses on large bodies of text, which can be likened to test suites. Several researchers have developed similarity metrics to compare ranked lists that are output from an IR query, compare documents, and compare a query to a document. This research in these areas is discussed in Section 2.4.1. Further, research on similarity between objects has been accomplished in software security to detect viruses and in neurocomputing to determine where to place an object in a fuzzy lattice. Being the only other software research on similarity found in the literature, this is discussed as relevant work in Section 2.4.2.

2.4.1 Similarity Metrics in IR and NLP

Previous work in similarity, specifically the work by Aslam and Frost, can be likened to determining how similar test cases may be based on the events they

contain, where *features* in IR map to *events* in software testing. Kilgarriff's work in comparing document collections, while different from test suite comparison, is very applicable to the research presented here. Just as is true for events in EDS test cases, Kilgarriff found that words in a corpus do not appear in a random order and techniques to compare them must take context into account.

A common problem in IR is correlating two ranked lists, often the output of an information retrieval query. According to Yilmaz *et al.*it is important to be able to weight a list, and usually more important to determine the difference between the high-ranking items in the list than to determine the difference in the low-ranking items [64]. A common solution is to use Kendall's τ statistic; however, Kendall's τ does not distinguish between the high-ranking and low-ranking items in the list. Another common approach is to use Spearman's correlation coefficient; this also has drawbacks in comparison. Yilmaz *et al.*, have proposed a statistic which extends Kendall's τ and gives more weight to the high-ranking items in the list. Other approaches to comparing two ranked lists can be found in work by Shieh, Haveliwala *et al.*, (the Kruskal-Goodman τ statistic), and Fagin *et al.*, (specifying the top-k of a list which should be treated differently) [27].

Aslam and Frost developed a similarity metric for documents, as an extension of work by Dekang Lin in object similarity [1]. Lin *et al.*'s metric is designed to compare documents based on the features contained in that document, from some possible *feature set* which is contained in the set of documents. Aslam and Frost extend the metric to compare normalized documents, thereby accounting for fractional features that would otherwise be lost during normalization. They found their

metric outperformed other standard metrics when run on a standard query retrieval data set.

In his introduction to the special issue of the Journal of Computational Linguistics: "The Web as a Corpus," Kilgarriff notes the problem of being able to come to a stable conclusion about a single word, and that it is often not possible to draw a conclusion about a combination of words or a rare word from a corpus [27]. In his work, Kilgarriff found that a method of comparison based on the χ^2 test is the most effective, followed closely by the Mann-Whitney ranks test [26]. He focused on determining both how similar two corpora are and in what ways two corpora differ, by determining which words are the most distinctive. Kilgarriff considered Poisson mixtures, Katz's model for word distributions, and adjusted frequencies used in research by Francis and Kučera. Finally, he tried various methods of comparing two corpora and determined several things: the distribution of word frequency is of little help; Spearman's rank correlation is too affected by the frequency of words that may not be important (such as *the*); and using a χ^2 test while ignoring the null hypothesis is a reasonable method of comparing corpora.

2.4.2 Similarity in Software

Previous research in similarity in the software engineering field is presented here. Karnik *et al.*'s work in virus detection, comparing source files as sequences of machine instructions, uses a very similar approach to that presented in this research [24]. However, while Karnik *et al.*, treat machine instructions in a different

sequence as functionally the same, this is not the case in EDS testing where event order is specific.

Research in virus detection has focused on examining method signatures for changes. Virus writers are aware of this, however, and have found ways to evade this check [24]. Using the knowledge that malicious software shares significant amounts of code, Karnik *et al.*, used the statistical properties of the morphed viruses to detect variants. Although virus variants may differ in the sequence of instructions they contain, they are functionally the same. To determine the similarity between functions, Karnik *et al.*'s method flagged any functions which are similar using a threshold value of 0.97, and then took the average of the similarities to evaluate overall program similarity. Karnik *et al.*, compared each function in A to each function in B and maintained an array with the cosine similarity measure for each comparison. By computing the cosine similarity between two versions, they were able to identify files with viruses [24].

Cripps and Nguyen proposed using cosine similarity measures as the inclusion measure used by fuzzy lattice neurocomputing (FLN) [11]. In this domain, data items of different types may be stored in the same lattice. Their work used the *attributes* of each data item to generate a vector which represents that item. These vectors can then be compared using the cosine similarity measure by computing the cosine of the angle between the two vectors. Extending this measure to a weighted attributed cosine similarity measure is done by introducing a constant into the computation, which provides a count, or weight, of that attribute for that data item.

Only one previous work in software testing has compared *test cases* within a suite to determine redundancy inside the suite [31]; a suite-to-suite comparison has not been performed. Traditional approaches for test suite minimization usually rely on coverage information collected during dynamic execution. Li *et al.*, applied a static analysis technique to detect redundant test cases based on their instruction sequences and counts.

2.5 Summary

The works presented in this chapter covered topics broadly related to this research: software testing, defect classification and similarity metrics. Previous research in GUI testing used a graph model to model the GUI and generate test cases. The research described in this document also uses a graph model, although the model is annotated in a completely new way, with conditional probabilities computed based on event sequences of length-n in existing test suites. The concepts of test suite reduction are broadly related to the concept of consolidation developed as part of this research; however, the goal of test suite reduction is different from that of consolidation.

Although many defect classification studies have been developed, none have focused on GUI defects and the differences between the GUI code and non-GUI code in an application. This research used a modified version of Beizer's defect taxonomy which includes a category for GUI defects.

Due to the need to determine how similar a result string is to a query string in

IR, and thereby improving a search engine's results, research in similarity metrics is very common in that field. NLP researchers also have strong motivation to improve their results in language translation, and use similarity metrics to determine how close their translation is the correct translation. However, in software engineering in general, and software testing specifically, similarity metrics are not often used. The ideas that have been successful in other research communities were used in this research to develop a test suite similarity metric.

Chapter 3

Studying the Effectiveness of Crash Testing for GUIs

In Chapter 1, several types of GUI testing were described, to include crash testing. The results on open source applications have been promising, showing crash testing to be useful. However, past researchers have not studied the efficacy of crash testing on industrial applications, and have often noted this limitation as a threat to validity.

As part of determining the effectiveness of crash testing, it is necessary to learn more about the characteristics of GUIs, such as the percentage of the system which is GUI code, the types of tests that are run to test through the GUI compared to the tests run to test the functionality of the GUI, and the types of defects found through crash testing. Therefore, this chapter will use industrial applications to further the knowledge of these aspects of GUIs.

More specifically, fielded, industrial applications produced and sold by ABB Group[1] were examined. Previous defect studies conducted at ABB [47] have had practical implications – they have been beneficial to developers, testers and managers by associating defects with particular phases in the software development process. Test teams have seen dramatic increases in the number of defects they are now able to detect in early phases of testing, resulting in significant decreases in low-level bug fixes, which were previously common in late-phase test.

[1] http://www.abb.com/

3.1 Empirical Study

The goal of this study is to *improve the overall quality of GUI testing by characterizing GUI systems using data collected from defects, test cases and source code to assist testers and researchers in developing more effective test strategies.* Using the Goal Question Metric (GQM) Paradigm [2], the goal is restated as follows:

> Analyze the **defects, test cases, and source metrics**
> for the purpose of **understanding**
> with respect to **GUI systems**
> from the point of view of the **tester/researcher**
> in the context of **industry-developed GUI software.**

This research goal is broken into four research questions to be answered by the study:

RQ3.1 How many defects in GUI applications are detected through crashes, as compared to observed program deviations?

RQ3.2 How do defects in the GUI differ as compared to overall defects in the AUT? What kinds of defects are commonly found through the GUI?

RQ3.3 How do GUI components compare to non-GUI components with respect to source metrics? How do source changes in GUI components compare to changes in non-GUI components?

RQ3.4 What are the characteristics of the test suites? How many of the tests are testing the GUI compared to testing the application through the GUI?

In determining the study setting, several key factors were decided, such as the defect taxonomy that would be used, how construction metrics could be used

to characterize GUI systems, and how test results could be leveraged for the GUI characterization. Each of these factors is described in the following sections.

3.1.1 Choosing and Tailoring a Taxonomy

Beizer's taxonomy [3] divides defects into eight main categories, each describing a specific set of defects. These are shown in Table 3.1. Each main category is further refined into three levels of subcategories. A defect is then assigned a four digit number with each digit representing the selected category or subcategory. For example, Processing Bugs would be 32xx, where the 3 designates a structural defect and the 2 places this defect into the processing subcategory. The last two numbers, shown as x here, refine the defect to more levels of detail. Beizer's taxonomy includes four levels of categories for each defect.

The first main category of the taxonomy is for **Functional defects**, *i.e.*, errors in the requirements, including defects caused by incomplete, illogical, unverifiable, or incorrect requirements. The second main category, **Functionality as Implemented**, deals with defects where the requirements are known to be correct but the implementation of these requirements was incorrect, incomplete, or otherwise wrong. The next two main categories, **Structural Defects** and **Data Defects**, are used for low-level developer defects in the code, such as problems with control flow predicates, loop iteration and termination, initialization of variables, incorrect types, and incorrect manipulation of data structures. Another main category of defects classifies **Implementation** errors. These are errors dealing with simple ty-

1xxx	Functional Bugs: Requirements and Features 11xx Requirements Incorrect 12xx Logic 13xx Completeness 14xx Verifiability 15xx Presentation 16xx Requirements Changes
2xxx	Functionality As Implemented 21xx Correctness 22xx Completeness Features 23xx Completeness Cases 24xx Domains 25xx User Messages and Diagnostics 26xx Exception Conditions Mishandled
3xxx	Structural Defect 31xx Control Flow and Sequencing 32xx Processing
4xxx	Data Defect 41xx Data Definition, Structure, Declaration 42xx Data Access and Handling
5xxx	Implementation Defect 51xx Coding and Typrgraphical 52xx Standards Violations 53xx GUI Defects 54xx Software Documentation 55xx User Documentation
6xxx	Integration Defect 61xx Internal Interfaces 62xx External Interfaces 63xx Configuration Interfaces
7xxx	System and Software Architecture Defect 71xx OS 72xx Software Architecture 73xx Recovery and Accountability 74xx Performance 75xx Incorrect diagnostic 76xx Partitions and overlays 77xx Environment 78xx 3rd Party Software
8xxx	Test Definition or Execution Bugs 81xx System Setup 82xx Test Design 83xx Test Execution 84xx Test Documentation 85xx Test Case Completeness

Table 3.1: Beizer's Taxonomy (Modified)

pographical errors, standards, or documentation. The next main category is for **Integration defects**, representing errors in the internal and external interfaces in the software. Finally, the last two main categories of defects deal with **System** and **Test defects**. System defects comprise errors in the architecture, OS, compiler, and failure recovery of the AUT. Test defects represent errors found in the test descriptions, configurations, and test programs used to validate the system.

Beizer's taxonomy was chosen for this study, based primarily on the categories themselves and the fit within the ABB environment. Currently, testing at ABB is not based on a test strategy, and therefore ODC was not chosen since it relies on a test strategy and process being in place. Other object oriented taxonomies were not chosen since the development of these applications is not strictly object oriented, although the language has the capability.

After selecting Beizer's taxonomy, all of the categories and subcategories were analyzed. A two level approach was selected, with only the main category and one subcategory used, due to the needs of ABB. Researchers at ABB tailored the taxonomy for initial work with developers and testers within ABB [47]. The main categories were kept; a few subcategories were renamed, giving them names more similar to those used inside ABB, and other subcategories were added for specific defect types due to their importance to ABB.

The first additional subcategory was named *GUI defects*, and assigned to the Implementation main category as 53, to categorize defects that exist either in the graphical elements of the GUI or in the interaction between the GUI and the underlying application API. These defects were given their own defect type since code involved in the GUI was treated differently than the underlying application code in many companies, and require different testing steps to validate.

The next change to the taxonomy involved dividing the documentation subcategory into two categories, one for classifying in-software documentation errors and one for user documentation errors. These were labeled 54 and 55, respectively. *In-software documentation* defects cover missing or incorrect developer documents,

such as design documents, or internal code documentation, *i.e.*, comments in the code. *User documentation* refers to defects that exist in documents that are released to the customer with the software, such as product installation and user manuals.

The taxonomy was further modified to include a subcategory to classify defects in *system setup*. This category allows classification of defects dealing with configuring the system correctly for its intended use. Since all of ABB's systems are highly configurable, these defects are important enough to track separately. This subcategory was added to the Test Definition or Execution Bugs category, and labeled as 81.

Finally, a subcategory was added to categorize defects in the *configuration interfaces* that are available in the system. Since these systems have so many possible executable configurations, each of which highly impact how the system executes, the interfaces which allow this configuration to occur require their own classification. This new defect type was added to the taxonomy as 63. The modified version of Beizer's taxonomy is shown in Table 3.1

3.1.2 Gathering Construction Metrics

In order to compare the construction of GUI and non-GUI components of the system, the source code files for one system were split into two groups: files implementing the interface of the system and files implementing the remaining functionality of the system. To aid in characterizing the systems, each of these groups was analyzed separately. Files were determined to be part of the GUI if they contained

code that implemented a GUI action, *i.e.*, a button click, menu click, window open or window hide event.

Source code metrics were collected using Source Monitor[2] for one AUT, and the results were divided into the GUI and non-GUI groups, based on the label of the corresponding file or class. Five source metrics were selected to represent measures of size, complexity, coupling, and developer documentation. These include lines of code (LOC), LOC per method, percentage of lines with comments, cyclomatic complexity (CC), and call depth. In addition to source code metrics, source code changes were derived by computing the difference for each metric between versions of the system. Although source code was only available for one AUT, it is the largest in the study, containing over 1.6 million lines of code.

3.1.3 Collecting Test Suite Data

The ABB product group defect repositories that were mined for this study do not uniformly contain information on whether or not the defect was detected through a crash. Therefore, to gather information on whether the system crashed to expose each defect, natural language queries were run on the text fields of the repositories, such as Title, Description, Evaluation, and Implementation Notes. Words used in the query include "access violation," "ACCESS_VIOLATION," "crash," "hang," "freeze," and "froze." This natural language query was reinforced by human data checking of the defect reports. Determining how many of the reported defects were detected through crashes provides evidence on the effectiveness of crash testing for

[2]http://www.campwoodsw.com/sourcemonitor.html

GUIs. It also assists in characterizing GUI systems.

Metrics pertaining to the test suites were also collected to determine characteristics of industrial test suites used for GUI systems. All three systems studied use manually executed tests, and one product augments this manual testing with a large suite of automated GUI tests. The manual test suite was only available for one of the AUTs and the automated tests were not available to include in this study. From the manual test suite and its results, metrics were gathered for the size of the test suite, the number of test cases used for crash testing, the method used to generate test suite (functional, logical, or state), and the number of tests with validation points. Validation points, points between test steps where the state of the application is checked, are often implied in manual testing, as the human tester can visually check the state of the system. For this study, validation points were determined from the test case and the results of the test case execution.

3.1.4 Subject Applications

The three applications chosen for this study were developed by ABB and are all Human Machine Interfaces (HMI) for large industrial control systems. They allow the user to monitor, configure, and control various aspects of the running system. These systems are developed in C++ and run on the Windows operating system. They were selected for this study because they are large, deployed applications that have been running in the field for over 10 years by hundreds of customers around the world. To protect company privacy, the systems will be referred to as AUT1,

AUT2, and AUT3 in this study.

3.1.5 Collecting GUI System Data

Defect data was manually mined from several repositories at ABB holding Software Problem Reports (SPRs). Specifically, the defects of interest are those found in late-phase testing and by customers after release. Each AUT had a separate repository. Two of the AUTs used the same repository while the third AUT used a different repository. However, all three repositories contain roughly the same data, and all of the data needed for this study was available for all of the systems.

Each SPR indicates when the defect was found, what version of the software was running, and the severity of the defect. Defect severity is assigned on a 5-point orthogonal scale ranging from Low to Project Stopper. For this study, 1,200 defects in the top three points of the severity scale were classified, since the management team has determined that the cost of discovering lower severity defects in the field can be tolerated.

Due to limited data query support in these repositories, the SPRs were saved in text files and parsed using a combination of manual effort (*i.e.*, members of the research team read the documents) and Perl scripts. After gathering the data into a central location and formatting the data consistently across the three AUTs, the analysis was conducted. The SPR data used for this study represents three years of development and two major versions of the products.

3.1.6 Threats to Validity

The results of this study should be considered with several possible threats to validity. First, the defect classification was performed by several people over the course of one year. Due to this method of classification, it is possible that the same type of defect was categorized differently by different people. However, to mitigate this risk, the team classifying the defects met each week and selected random groups to reclassify together. If issues were found, others with similar classifications were also discussed.

Second, crash data was mined from the defect repositories using natural language queries. These queries pose a risk of missing data due to the imprecise nature of language. To decrease the number of missed crashes, the queries included the synonyms and several misspellings of each search term. In addition, a set of random groups of defects were selected and one of the authors manually determined if the defect was a crash or not. If an uncaught crash was detected, additional keywords were added to the query and it was rerun. The crash results were also checked for false positives, but none were found.

Third, the defects analyzed for this study are from large, currently deployed production systems. While they are applicable to a variety of domains, they are primarily control and monitoring systems and therefore the results may not be directly transferrable to systems in other domains, such as desktop or productivity applications.

3.2 Results

From the original goal presented in Section 3.1, a set of research questions was developed for this study. Each research question has an associated set of metrics that were collected to provide insight into the problem. These metrics, and their values, are presented here, along with the research question to which they apply.

3.2.1 Overall Defects

RQ3.1: How many defects in the GUI applications were detected through crashes, as compared to observed program deviations?

Metrics: *Number of defects detected by software crash, number of defects detected by observed program deviations*

Since it was possible to use a primarily automated method for determining crashes, a total of 3,869 defects were analyzed, encompassing all five severity types and all three AUTs. Processing the natural language query described in Section 3.1.3 produced the following crash results: AUT1 had 248 crashes out of 1,661 defect reports; AUT2 had 372 crashes out of 1,892 defect reports; and AUT3 had 37 crashes out of 316 defect reports. Therefore, crashes accounted for 15%, 19% and 12%, respectively, of the defects detected.

RQ3.2: How do defects in the GUI differ as compared to overall defects in the AUT? What kinds of defects are commonly found through the GUI?

Metrics: *Defect classification by type, software lifecycle phase of defect detection*

For this research question, GUI defects were processed manually. 1,215 defects

	Number of Defects Found	
Application	GUI	Non-GUI
AUT1	7	154
AUT2	11	727
AUT3	90	226

Table 3.2: GUI and Non-GUI Defects per AUT

from three different GUI systems were studied and classified into the taxonomy. This list only includes defects with a severity of High, Critical, and Project Stopper (the top three categories on a 5-point scale) that were found in late testing phases or by customers in the field. Since these defects are the highest priority for ABB managers, these represent the defects that are most often fixed and included in later releases. Table 3.2 shows the number of the GUI and non-GUI defects found in the systems. The combined classification data for these defects is shown in Table 3.3, ordered by defect rank. The most common defects were in data access and handling (15.47%) and control flow and sequencing (12.67%). Out of the 27 defect classes, the top 4 classes accounted for approximately 50% of the defects. GUI defects ranked fifth overall.

3.2.2 Construction

RQ3.3: How do GUI components compare to non-GUI components with respect to source metrics? How do source changes in GUI components compare to changes in non-GUI components?

Metrics: *File changes, average statements per method, number of statements changed, number of lines changed, percentage of commented lines, average complexity, average block depth*

Defect Class	Fault Type	% Defects
42	Data Access, Handling	15.47%
31	Control Flow, Sequencing	12.67%
21	Correctness	11.69%
32	Processing	10.37%
53	GUI	8.89%
81	System Setup	5.76%
23	Part. Implemented Features	4.77%
41	Data Def., Struc, Decl.	4.53%
72	Software Architecture	3.95%
22	Unimplemented Features	3.62%
26	Error Handling, Missing, Incorrect	3.37%
25	User Messages and Errors	2.63%
61	Internal Interfaces	2.63%
55	User Documentation	1.98%
75	Third Party Software	1.48%
71	OS and Compiler	1.23%
54	In-Software Documentation	0.82%
74	Performance	0.82%
62	External Interfaces	0.74%
24	Domains	0.66%
51	Coding and Typological	0.58%
83	Test Execution	0.33%
63	Configuration Interfaces	0.25%
73	Recovery	0.25%
82	Test Design	0.25%
16	Requirements Changes	0.16%
52	Standards Violation	0.08%

Table 3.3: Defect type across all systems

Construction was investigated for one of the AUTs, due to the availability of its code. After dividing the source code into two groups, the GUI and non-GUI portions of the system (Section 3.1.2), their source metrics were calculated. The two groups of measures were compared using a two sample student t-test assuming unequal variances to compare the means of the two groups. The hypothesized difference in means was zero. This test was selected since the number of observations to compare was approximately 15,000. The results of the source metrics analysis shows that

there are statistically significant differences (at $\alpha = 0.05$) between the GUI and non-GUI components for all five of the metrics selected ($p < 0.05$). Table 3.4 contains the computed metrics for the system used in this part of the study.

		Mean	StdDev
LOC	GUI	388.27	516.49
	Non-GUI	248.33	14542.59
LOC / Method	GUI	13.10	6.77
	Non-GUI	3.57	8.93
% Comments	GUI	11.54	11.92
	Non-GUI	20.40	17.09
Complexity	GUI	3.58	1.63
	Non-GUI	2.05	5.01
Depth of Call Tree	GUI	1.47	0.45
	Non-GUI	0.82	0.72

Table 3.4: Source code metrics

Table 3.5 shows the code changes in the GUI and non-GUI parts of the system for five versions of the AUT. The table shows that the mean of the number of statement changes between versions for GUI and non-GUI are similar, but the standard deviation is significantly larger for the non-GUI parts of the system.

	GUI		Non-GUI	
Version	Mean	StdDev	Mean	StdDev
V1 - V2	77.17	119.27	67.32	1016.74
V2 - V3	77.92	150.01	94.70	2251.50
V3 - V4	38.88	76.54	61.57	948.73
V4 - V5	15.36	73.33	16.28	170.40

Table 3.5: Statement changes

Figure 3.1 shows the percentage of change to the GUI and non-GUI portions of the system for the five versions studied. On average, 8% of the changes were to GUI portions and 92% were to non-GUI portions of the system. The overall size of the system is approximately 1.6 MLOC, of which the GUI portion of the

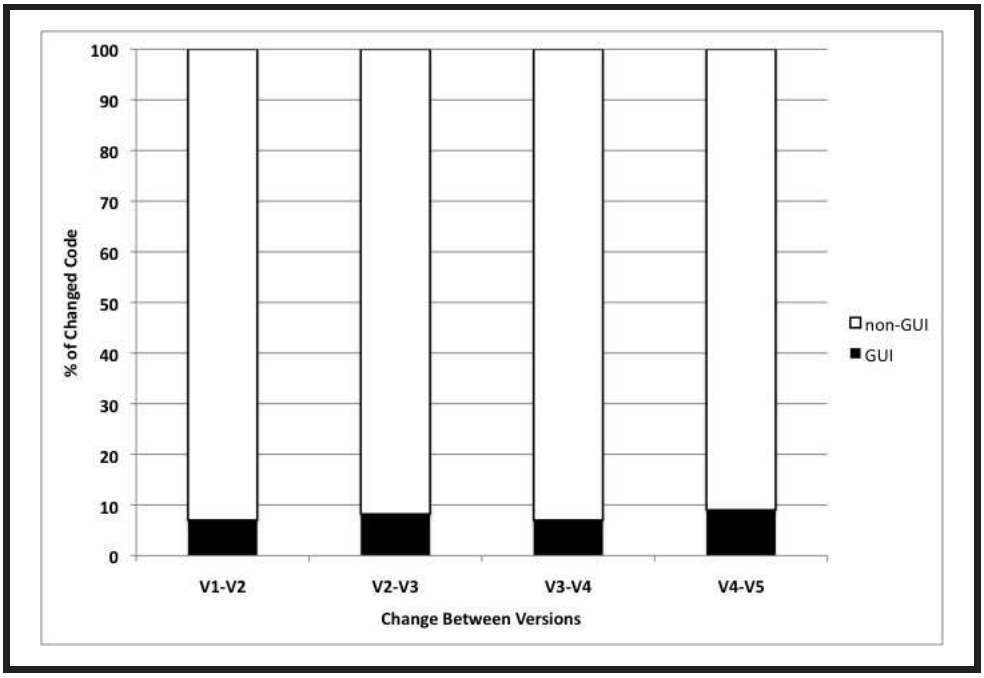


Figure 3.1: Changes to GUI and non-GUI portions

system contains approximately 200 KLOC (14%) and the non-GUI portion contains approximately 1.4 MLOC (86%).

3.2.3 Test Suites

RQ3.4: What are the characteristics of the test suites? How many of the tests are testing the GUI compared to testing the application through the GUI?

Metrics: *Number of test cases with validation points, number of test cases used for crash testing, size of test suite, method used to generate test suite (functional, logical, state)*

For this study, the current product testing suite for one of the AUTs was analyzed. This suite is executed manually and takes approximately three man-weeks to complete. It contains 800 test cases in total. Of these, 42% contain

specific verification points, 50% are only looking for crashes, and the remaining 8% contain general statements of what the correct behavior should be. 20% of the tests were designed to test the GUI itself and the remaining 80% were designed to test the application through the GUI. The main testing methods used in the suite include creating tests for the general cases (80%), error cases (8%), boundary values (10%), and state or combinatorial testing (2%).

3.3 Discussion

This section presents the analysis of the metrics data shown in the previous section and provides insight into the data gathered for the research questions posed for this study.

3.3.1 Overall Defects

The defect data presented in Section 3.2.1 provides an industry defect profile of three large deployed GUI applications. Table 3.3 provides a distribution of defect types that were found through the GUI during late phase testing and after deployment. A large portion of the defects found were categorized as *42: data access and handling*, *31: control flow and sequencing*, *21: correctness*, and *32: processing*, representing 50% of the total defects.

Defects in the GUI itself represented only 6% of the total defects found across the three systems. Looking at the individual systems, Table 3.2 shows that less than 5% of the defects for two systems were GUI defects, while the third system

had almost 30%. Due to the fact that all of these defects were detected through the GUI, it is surprising that more of the defects are not related to the GUI. Many of the problem reports studied during data collection described incorrect system behavior that was observed through the GUI. However, the actual defect often resided in the underlying system components rather than the GUI itself. The assumption that the problem would be rooted in the GUI was primarily due to the limited observability into the system that the GUI provides.

Upon examination of the defects detected in the systems, most required observability of program deviations and a knowledge of the expected behavior of the systems. Conversely, few defects resulted in a crash of the running systems. This also highlights the need for good observability into the system when it is tested through its GUI.

3.3.2 Construction

The results shown in Section 3.2.2 indicate that there is a significant difference in the source code metrics when the GUI and non-GUI components of the system are compared. On average, GUI components are larger than their non-GUI counterparts. The percentage of the code that is commented in the GUI components of the system is much lower than that of the non-GUI components. The GUI code is nearly two times more complex than the non-GUI code. Finally, the depth of the call tree is much larger in the GUI portions of the code.

These four measures, taken together, may indicate that developers do not

create GUI components with the same discipline that they use when creating the rest of the system. GUI components also contain glue code which links the GUI events to their respective API calls in the underlying system. This extra step may cause some of the extra size, complexity, and call depth.

Section 3.2.2 also investigates the difference in source code changes between the GUI and non-GUI portions of the system. The results show that the average number of changes is similar between GUI and non-GUI components. However, there are significantly more non-GUI components, leading to a much larger overall number of changes to the non-GUI portion of the system, shown in Figure 3.1.

3.3.3 Test Suites

The data presented in Section 3.2.3 shows that the majority of the test cases (80%) are intended to test the application through the GUI, while the remaining test cases (20%) are intended to test the GUI itself. Test suites were also characterized on the absence or presence of verification points in the test cases. In the suites studied, 50% of the written tests are verified solely on whether or not the system crashed when the test case was executed. 42% of the tests contained specific criteria for the tester to verify through the GUI that the system performed as expected. Finally, upon examining characteristics of the test suite, it was determined that most of the test suite was executing test cases solely based on the general case of the system (80%), seldom applying additional test methodologies such as boundary checking (10%), state-based testing (2%) and checking error conditions (8%), methods seen

as good testing practices.

3.4 Conclusions

This chapter presented the results of a study characterizing GUI systems and their test suites to add to the knowledge base for testers and researchers alike as they determine the efficacy of crash testing for GUI systems. Traditionally, GUI testing has relied on crash testing, due to several factors, including difficulty in developing test suites that adequately cover the breadth and depth of the system as well as the need to observe the underlying system's behavior [61].

The findings on the test suites could be due to employing the strategy of testing the application through the GUI rather than testing the business logic of the application separately from the GUI. As testers are focused on the GUI, and the observability into the system that it provides, the ability to understand and verify the behavior of the underlying system is compromised. This lack of observability into the system often inhibits the testers from using additional test design methodologies, such as testing boundary conditions and checking error conditions, since that level of observability is not available.

Next, Chapter 4 will describe the CONTEST algorithm and model. CONTEST uses a new probabilistic GUI model that can be generated based on information gleaned from existing test suites. This model conserves the context of the existing event sequences and is used to consolidate the original test suites, while conserving the fault detection effectiveness and code coverage of the original suites.

Chapter 4

CONTEST: An N-Gram Model for Consolidation of Sequence-based GUI Test Cases

This chapter describes CONTEST, a method of modeling a GUI based on the n-gram model often used in NLP, that can be encoded with existing test suites, and further used to generate a new, consolidated test suite. The model maintains the context of each event executed in the test suite, for each length-n subsequence. Probabilities based on the likelihood of each event sequence guide the test suite consolidation. An empirical study using crash testing was conducted to show the effectiveness of CONTEST. Recall that Chapter 3 showed that crash testing is a useful and valid technique for finding defects in GUI applications.

This chapter provides concrete examples of (a) why CONTEST was able to generate new sequences that detected new faults, (b) why CONTEST eliminated some sequences that had previously detected faults, causing the new consolidated suite to miss some of the faults, (c) why CONTEST covered new code, and (d) why CONTEST missed some code previously covered by the original suites are given. For each of these cases, the relevant parts of the n-gram model, the probabilities along its nodes/edges, and how they were handled by CONTEST's test case generator are shown.

4.1 Modeling the GUI for CONTEST

The core enabler of CONTEST is a probabilistic model of the GUI. Building upon the concepts summarized in Section 2.1.1, this section describes the probabilistic GUI model used by CONTEST. Extending the notion of a tailored EFG, multiple event sequences may be represented in a single tailored EFG. For example, the 13 event sequences shown in Figure 1.2 that were used in the example in Chapter 1, $< e_1, e_4 >$, $< e_1, e_6 >$, $< e_1, e_3, e_5, e_6 >$, $< e_2, e_6 >$, $< e_2, e_3, e_5 >$, $< e_2, e_3, e_5, e_6 >$, $< e_2, e_4, e_7, e_2, e_6 >$, $< e_3, e_5, e_4, e_3, e_6 >$, $< e_7, e_1 >$, $< e_7, e_1, e_6 >$, $< e_7, e_2, e_6 >$, $< e_7, e_2, e_3 >$, and $< e_7, e_2, e_3, e_5, e_6 >$, may be used to create the tailored[1] EFG shown in Figure 4.1.

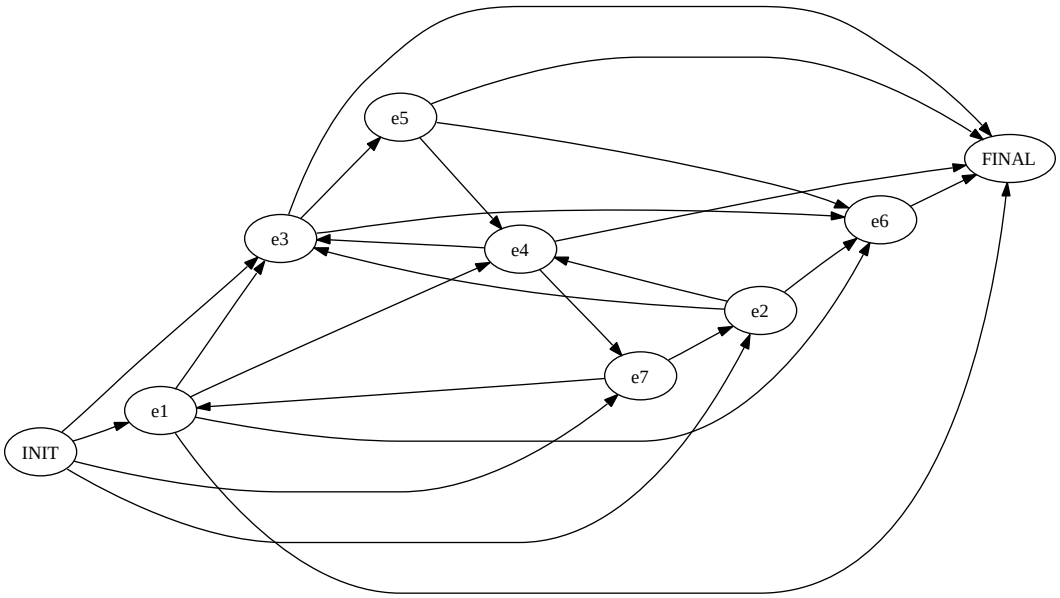


Figure 4.1: EFG for the Running Example

Without loss of generality, the EFG is augmented with two special nodes, $INIT$ and $FINAL$. The event $INIT$ has an edge to the first event of each event

[1]The term "tailored" is dropped for the remainder of this document.

sequence; the event $FINAL$ has an edge from the last event in each event sequence. This type of EFG is the basis for test-case generation in CONTEST.

To use the EFG to consolidate the original sequences, it can be annotated with probabilistic relationships between events. Consider the EFG of Figure 4.1 and the 13 input sequences. From the EFG, it can be seen that only e_1, e_2, e_3, and e_7 may be the first event in a test case (because there is an edge from $INIT$ to each of these events). Upon examination of the 13 sequences, it can be seen that e_1 starts 3 sequences, e_2 starts 4 sequences, e_3 starts 1 sequence, and e_7 starts 5 sequences. When generating consolidated test suites from this EFG, ideally, the algorithm should ensure that the percentage of test cases in the consolidated suite reflects the distribution in the original suites. Therefore, most of the test cases (perhaps $\frac{5}{13} \times 100$) start with e_7, $\frac{3}{13} \times 100$ start with e_1, $\frac{4}{13} \times 100$ start with e_2, few or $\frac{1}{13} \times 100$ start with e_3, and very few or none with e_4, e_5, or e_6.

Moreover, from the EFG, it can be seen that e_1 and e_2 can follow e_7. The subsequence $< e_7, e_2 >$ appears 4 times in the input sequences while $< e_7, e_1 >$ appears only twice. With the goal of consolidating test suites in mind, test case generation must ensure that $< e_7, e_2 >$ appears more often, at least twice as often, as a subsequence than $< e_7, e_1 >$.

Another interesting point to note from the original event sequences is that of the four times e_2 follows e_7, it does so thrice in the context of e_7 being the first event; only once does it do so when e_7 follows e_4. This seems to suggest that in order to obtain a resulting test suite that is "similar" to the original suite, it would be desirable to consider a *history* of several previous events. Therefore, the algorithm

will compute the probability of e_2 following e_7 *given that* e_7 has followed e_4.

Consider the annotated EFG shown in Figure 4.2. Each node (corresponding to an event e_x) in the original EFG is associated with a conditional probability table. The table has two columns; the first column is a subsequence $< e_a, e_b >$ of length 2; the second column is the probability that e_x follows $< e_a, e_b >$ in the input event sequences. For example, consider the table for e_2. Because only e_2 follows the subsequence $< e_4, e_7 >$, the probability entry is 1.0. On the other hand, e_2 follows $< INIT, e_7 >$ 3 times in the original sequences while e_1 follows $< INIT, e_7 >$ 2 times. Therefore, the probability entry of e_2 in the context of $< INIT, e_7 >$ is 0.6; e_1 in the context of $< INIT, e_7 >$ is 0.4.

Intuitively, the annotated EFG can be used for test case generation by starting with the $INIT$ event and traversing high-probability paths until the $FINAL$ event is reached. Next, an explanation of how the conditional probability tables are computed and a formal description of the test case generation algorithm are provided.

Developing the probabilistic EFG for the CONTEST algorithm starts by considering the input sequences as paths in the annotated EFG. In the running example, the paths are as follows:

$$r_1 = INIT, e_1, e_4, FINAL$$

$$r_2 = INIT, e_1, e_6, FINAL$$

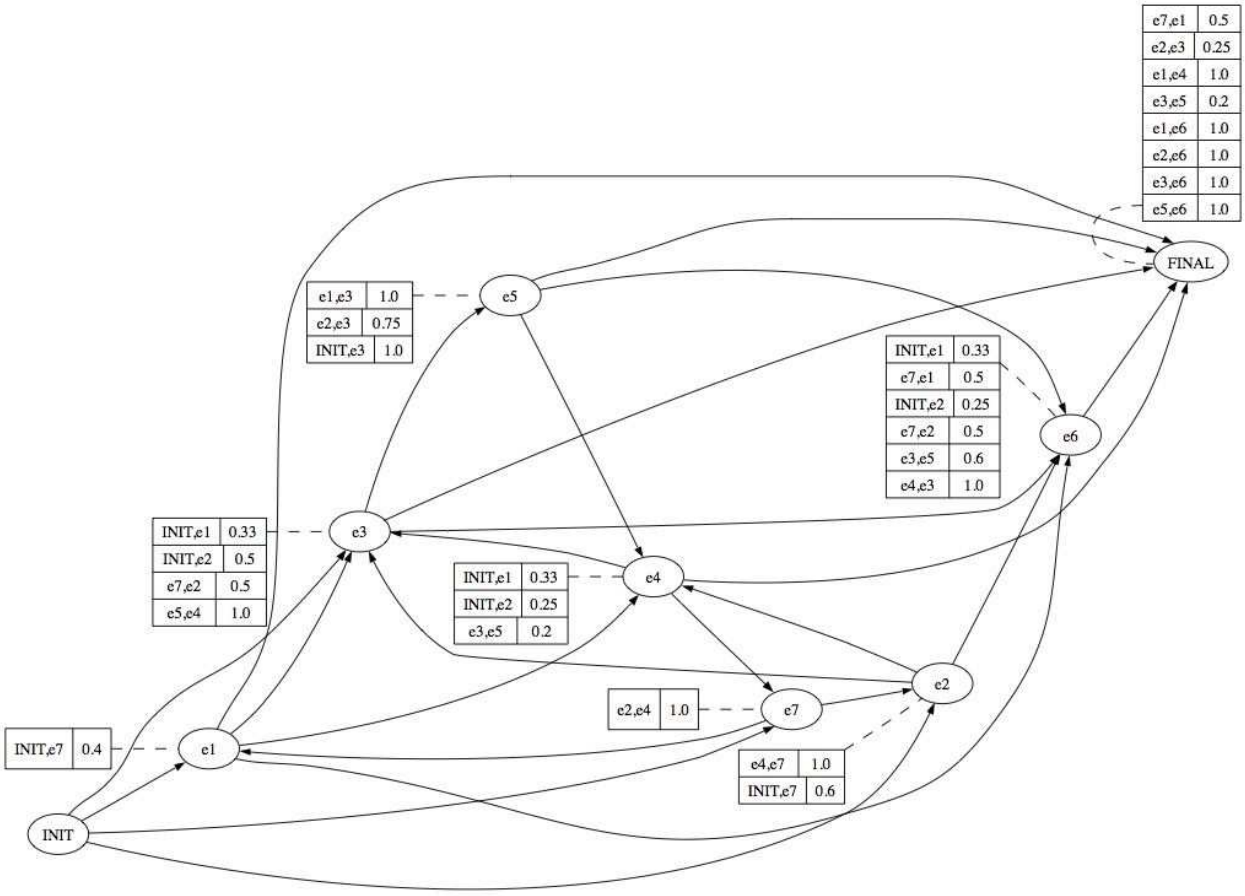


Figure 4.2: Annotated EFG for $H = 2$

$$r_3 = INIT, e_1, e_3, e_5, e_6, FINAL$$

$$r_4 = INIT, e_2, e_6, FINAL$$

$$r_5 = INIT, e_2, e_3, e_5, FINAL$$

$$r_6 = INIT, e_2, e_3, e_5, e_6, FINAL$$

$$r_7 = INIT, e_2, e_4, e_7, e_2, e_6, FINAL$$

$$r_8 = INIT, e_3, e_5, e_4, e_3, e_6, FINAL$$

$$r_9 = INIT, e_7, e_1, FINAL$$

$$
\begin{aligned}
r_{10} &= INIT, e_7, e_1, e_6, FINAL \\
r_{11} &= INIT, e_7, e_2, e_6, FINAL \\
r_{12} &= INIT, e_7, e_2, e_3, FINAL \\
r_{13} &= INIT, e_7, e_2, e_3, e_5, e_6, FINAL
\end{aligned}
$$

Definition: The prior probability that a randomly selected event from any of $r_1, r_2, \ldots, r_R$ is e_i is:

$$P(e_i) = \frac{count(e_i)}{\Sigma_{j=1}^{E} count(e_j)},$$

where $count(e_i)$ returns the number of times event e_i occurs in all paths $r_1, r_2, \ldots, r_R$ and E is the set of all events, including $INIT$ and $FINAL$.

Continuing with the running example, the counts and probabilities for each event are:

$$
\begin{aligned}
count(e_1) = 5 \qquad & P(e_1) = 0.08 \\
count(e_2) = 7 \qquad & P(e_2) = 0.10 \\
count(e_3) = 7 \qquad & P(e_3) = 0.10 \\
count(e_4) = 3 \qquad & P(e_4) = 0.05 \\
count(e_5) = 5 \qquad & P(e_5) = 0.08 \\
count(e_6) = 8 \qquad & P(e_6) = 0.12 \\
count(e_7) = 5 \qquad & P(e_7) = 0.08 \\
count(INIT) = 13 \qquad & P(INIT) = 0.20 \\
count(FINAL) = 13 \qquad & P(FINAL) = 0.20
\end{aligned}
$$

Now, $count(e_i)$ and the prior probability calculation are extended from single events to sequences of events. Let s be a length-S subsequence of events in the input event sequences. For a subsequence length of 2 (*i.e.*, $S = 2$), the valid subsequences for this example are: $s_1 =< e_1, e_3 >, s_2 =< e_1, e_4 >, s_3 =< e_1, e_6 >, s_4 =< e_2, e_3 >, s_5 =< e_2, e_4 >, s_6 =< e_2, e_6 >, s_7 =< e_3, e_5 >, s_8 =< e_3, e_6 >, s_9 =< e_4, e_3 >, s_{10} =< e_4, e_7 >, s_{11} =< e_5, e_4 >, s_{12} =< e_5, e_6 >, s_{13} =< e_7, e_1 >, s_{14} =< e_7, e_2 >, s_{15} =< INIT, e_7 >, s_{16} =< INIT, e_3 >, s_{17} =< INIT, e_1 >, s_{18} =< INIT, e_2 >, s_{19} =< e_1, FINAL >, s_{20} =< e_3, FINAL >, s_{21} =< e_4, FINAL >, s_{22} =< e_5, FINAL >, s_{23} =< e_6, FINAL >$.

The next step in generating the probabilistic EFG is to compute the prior probability for each of these subsequences.

Definition: The prior probability that a randomly selected, length-S subsequence from any of $r_1, r_2, \ldots, r_R$ turns out to be s is

$$P(s) = \frac{count(s)}{\Sigma_{s_i \in subs(S)} count(s_i)},$$

where $count(s)$ returns the number of times s occurs as a subsequence of $r_1, r_2, \ldots, r_R$ and $subs(S)$ is the set of all length-S subsequences in $r_1, r_2, \ldots, r_R$.

For each subsequence of length 2 given above, the count and probability are:

$$count(s_1) = 1 \qquad P(s_1) = 0.2$$

$$count(s_2) = 1 \qquad P(s_2) = 0.2$$

$$count(s_3) = 2 \qquad P(s_3) = 0.4$$

$$count(s_4) = 4 \qquad P(s_4) = 0.5$$

$$count(s_5) = 1 \qquad P(s_5) = 0.125$$

$$count(s_6) = 3 \qquad P(s_6) = 0.375$$

$$count(s_7) = 5 \qquad P(s_7) = 0.71$$

$$count(s_8) = 1 \qquad P(s_8) = 0.14$$

$$count(s_9) = 1 \qquad P(s_9) = 0.33$$

$$count(s_{10}) = 1 \qquad P(s_{10}) = 0.33$$

$$count(s_{11}) = 1 \qquad P(s_{11}) = 0.2$$

$$count(s_{12}) = 3 \qquad P(s_{12}) = 0.6$$

$$count(s_{13}) = 2 \qquad P(s_{13}) = 033.$$

$$count(s_{14}) = 4 \qquad P(s_{14}) = 0.67$$

$$count(s_{15}) = 5 \qquad P(s_{15}) = 0.39$$

$$count(s_{16}) = 1 \qquad P(s_{16}) = 0.08$$

$$count(s_{17}) = 3 \qquad P(s_{17}) = 0.23$$

$$count(s_{18}) = 4 \qquad P(s_{18}) = 0.31$$

$$count(s_{19}) = 4 \qquad P(s_{19}) = 0.2$$

$$count(s_{20}) = 4 \qquad P(s_{20}) = 0.14$$

$$count(s_{21}) = 4 \qquad P(s_{21}) = 0.33$$

$$count(s_{22}) = 4 \qquad P(s_{22}) = 0.2$$

$$count(s_{23}) = 4 \qquad P(s_{23}) = 1.0$$

Next, it is necessary to compute the conditional probability of each subsequence followed by any event that follows it.

Definition: The conditional probability of each subsequence $s \in subs(S)$ that

immediately precedes e_i is computed as follows:

$$\forall s \in subs(S), \forall e_i \in E : P(e_i|s) = \frac{count(< s, e_i >)}{count(< s, e_x >)},$$

where $< s, e_i >$ denotes that e_i immediately follows s and $subs(S)$ is the set of all length-S subsequences in $r_1, r_2, \ldots, r_R$. Event e_x is a placeholder that can instantiate with any event that follows s.

These conditional probabilities are used to annotate the EFG. To illustrate how the probabilities shown in Figure 4.2 were computed, a step-by-step demonstration of some of the computations is given. First, to determine the probability of e_4 occurring after a sequence of 2 events, the set of events given as input will be examined for e_4 occurring after 2 events. This examination yields three possibilities: e_4 follows $< INIT, e_1 >$, $< INIT, e_2 >$, and $< e_3, e_5 >$. Therefore, three probabilities will be computed: $P(e_4| < INIT, e_1 >)$, $P(e_4| < INIT, e_2 >)$, and $P(e_4| < e_3, e_5 >)$. Before these can be computed, it must be determined which events other than e_4 follow the subsequences of interest, and the number of times that occurs must be counted. The sequence $< INIT, e_1 >$ is followed by e_3, e_4 and e_6, once in each case. The relevant computations follow:

$$P(e_3| < INIT, e_1 >) = \frac{count(< INIT, e_1, e_3 >)}{count(< INIT, e_1, e_x >)} = \frac{1}{3} = 0.33,$$

where e_x represents any event that follows $< INIT, e_1 >$. Likewise,

$$P(e_4| < INIT, e_1 >) = \frac{count(< INIT, e_1, e_4 >)}{count(< INIT, e_1, e_x >)} = \frac{1}{3} = 0.33$$

and

$$P(e_6| < INIT, e_1 >) = \frac{count(< INIT, e_1, e_6 >)}{count(< INIT, e_1, e_x >)} = \frac{1}{3} = 0.33$$

are computed.

The same process is used to compute $P(e_4| < INIT, e_2 >)$, and $P(e_4| < e_3, e_5 >)$. Subsequence $< INIT, e_2 >$ is followed by e_3 (twice), e_6, and e_4, therefore:

$$P(e_4| < INIT, e_2 >) = \frac{count(< INIT, e_2, e_4 >)}{count(< INIT, e_2, e_x >)} = \frac{1}{4} = 0.25$$

The subsequence $< e_3, e_5 >$ is followed by e_4, e_6 (thrice), and $FINAL$, therefore:

$$P(e_4| < e_3, e_5 >) = \frac{count(< e_3, e_5, e_4 >)}{count(< e_3, e_5, e_x >)} = \frac{1}{5} = 0.2$$

In all calculations of $P(e_i|s)$, the length of s is 2. The same formulae hold for other lengths of s. Consider length 3 sequences, (*i.e.*, $S = 3$); the valid subsequences and the $follows$ relationship for each subsequence are:

$s_{24} =< e_1, e_3, e_5 >$, $s_{25} =< e_2, e_3, e_5 >$, $s_{26} =< e_2, e_4, e_7 >$, $s_{27} =< e_3, e_5, e_6 >$,

$s_{28} =< e_3, e_5, e_4 >$, $s_{29} =< e_4, e_3, e_6 >$, $s_{30} =< e_4, e_7, e_2 >$, $s_{31} =< e_5, e_4, e_3 >$, $s_{32} =< e_7, e_1, e_6 >$, $s_{33} =< e_7, e_2, e_3 >$, $s_{34} =< e_7, e_2, e_6 >$, $s_{35} =< INIT, e_1, e_3 >$, $s_{36} =< INIT, e_1, e_4 >$, $s_{37} =< INIT, e_1, e_6 >$, $s_{38} =< INIT, e_2, e_3 >$, $s_{39} =< INIT, e_2, e_4 >$, $s_{40} =< INIT, e_2, e_6 >$, $s_{41} =< INIT, e_3, e_5 >$, $s_{42} =< INIT, e_7, e_1 >$, $s_{43} =< INIT, e_7, e_2 >$, $s_{44} =< e_1, e_4, FINAL >$, $s_{45} =< e_1, e_6, FINAL >$,$s_{46} =< e_2, e_3, FINAL >$, $s_{47} =< e_2, e_6, FINAL >$, $s_{48} =< e_3, e_5, FINAL >$, $s_{49} =< e_3, e_6, FINAL >$, $s_{50} =< e_5, e_6, FINAL >$, $s_{51} =< e_7, e_1, FINAL >$

Computing the prior probability for subsequences of length 3 yields the following:

$$count(s_{24}) = 1 \qquad P(s_{24}) = 0.06$$

$$count(s_{25}) = 3 \qquad P(s_{25}) = 0.18$$

$$count(s_{26}) = 1 \qquad P(s_{26}) = 0.06$$

$$count(s_{27}) = 3 \qquad P(s_{27}) = 0.18$$

$$count(s_{28}) = 1 \qquad P(s_{28}) = 0.06$$

$$count(s_{29}) = 1 \qquad P(s_{29}) = 0.06$$

$$count(s_{30}) = 1 \qquad P(s_{30}) = 0.06$$

$$count(s_{31}) = 1 \qquad P(s_{31}) = 0.06$$

$$count(s_{32}) = 1 \qquad P(s_{32}) = 0.06$$

$$count(s_{33}) = 2 \qquad P(s_{33}) = 0.12$$

$$count(s_{34}) = 2 \qquad P(s_{34}) = 0.12$$

$$count(s_{35}) = 1 \qquad P(s_{35}) = 0.33$$

$$count(s_{36}) = 1 \qquad P(s_{36}) = 0.33$$

$$count(s_{37}) = 1 \qquad P(s_{37}) = 0.33$$

$$count(s_{38}) = 2 \qquad P(s_{38}) = 0.5$$

$$count(s_{39}) = 1 \qquad P(s_{39}) = 0.25$$

$$count(s_{40}) = 1 \qquad P(s_{40}) = 0.25$$

$$count(s_{41}) = 1 \qquad P(s_{41}) = 1$$

$$count(s_{42}) = 2 \qquad P(s_{42}) = 0.4$$

$$count(s_{43}) = 3 \qquad P(s_{43}) = 0.6$$

$$count(s_{44}) = 1 \qquad P(s_{44}) = 1$$

$$count(s_{45}) = 2 \qquad P(s_{45}) = 1$$

$$count(s_{46}) = 1 \qquad P(s_{46}) = 0.25$$

$$count(s_{47}) = 3 \qquad P(s_{47}) = 1$$

$$count(s_{48}) = 1 \qquad P(s_{48}) = 0.25$$

$$count(s_{49}) = 1 \qquad P(s_{49}) = 1$$

$$count(s_{50}) = 3 \qquad P(s_{50}) = 1$$

$$count(s_{51}) = 1 \qquad P(s_{51}) = 0.5$$

Figure 4.3 shows the annotated EFG resulting from setting H to 3. To illustrate how the probabilities shown in Figure 4.3 were computed, a similar process to that given previously for Figure 4.2 is shown. To determine the probability of e_5 occurring after a sequence of 3 events, three probabilities will be computed: $P(e_5| < e_7, e_2, e_3 >)$, $P(e_5| < INIT, e_1, e_3 >)$, and $P(e_5| < INIT, e_2, e_3 >)$. Again, it is necessary to determine and count the events that follow the sequence $< e_7, e_2, e_3 >$; in this case, e_5 and $FINAL$, one time in each case. Therefore,

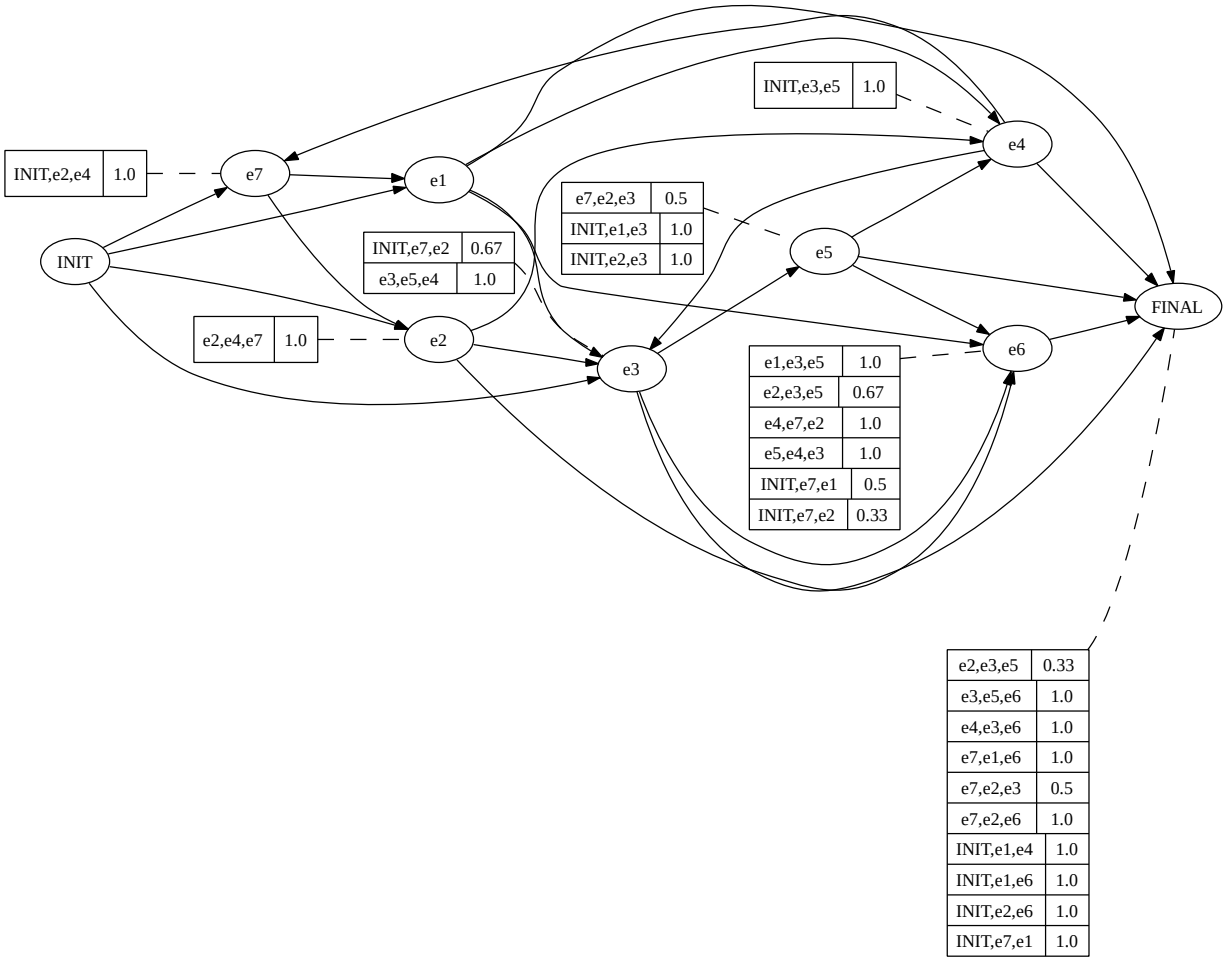


Figure 4.3: Annotated EFG for $H = 3$

compute

$$P(e_5| < e_7, e_2, e_3 >) = \frac{count(< e_7, e_2, e_3, e_5 >)}{count(< e_7, e_2, e_3, e_x >)} = \frac{1}{2} = 0.5.$$

Likewise,

$$P(FINAL| < e_7, e_2, e_3 >) = \frac{count(< e_7, e_2, e_3, FINAL >)}{count(< e_7, e_2, e_3, e_x >)} = \frac{1}{2} = 0.5$$

is computed. The same process is followed to compute $P(e_5| < INIT, e_1, e_3 >)$, and

$P(e_5| < INIT, e_2, e_3 >)$:

$$P(e_5| < INIT, e_1, e_3 >) = \frac{count(< INIT, e_1, e_3, e_5 >)}{count(< INIT, e_1, e_3, e_x >)} = \frac{1}{1} = 1.0$$

since $< INIT, e_1, e_3 >$ is only followed by e_5.

$$P(e_5| < INIT, e_2, e_3 >) = \frac{count(< INIT, e_2, e_3, e_5 >)}{count(< INIT, e_2, e_3, e_x >)} = \frac{2}{2} = 1.0,$$

also, since $< INIT, e_2, e_3 >$ is only followed by e_5.

The preceding example was provided to illustrate the effect of varying the length of s when computing the probabilities for the annotated EFG. In practice, testers will have to find a value of s that produces a consolidated test suite that meets their goals. Varying the length of s in this way creates two special cases. First, $P(e_i)$ can be thought of as $P(e_i|s)$ when s has length 0. This is not the same as $P(e_i|INIT)$, which is the probability that event e_i is the first event in the sequence, occurring immediately after $INIT$. Rather, $P(e_i|s)$ is the probability of e_i given *no information* about the events that precede it. Second, if S is equal to the length of the input sequences, the sequences output by the model will match the input and have a probability of 1.0.

The length of the subsequence s is called the *history* used by the n-gram model, denoted by H, and is given as a parameter to the model. Each set of entries for all length-h subsequences, $0 \leq h \leq H$, succinctly encodes a probabilistic Markov

model whose $O(E^h)$ nodes correspond to length-h subsequences and whose nodes are labeled with conditional probabilities.

4.2 Generating test cases

After the probabilistic model is generated, test cases will be generated that exercise the most probable sequence of events, given a range of starting points and test case lengths. All possible test cases are generated, based on the events included in the input set of event sequences, up to a specified history provided by the tester.

Before elucidating the details of the algorithm, the example sequences given in the previous section are used to illustrate an intuitive explanation of how test cases are generated. In the case of $H = 3$, it is necessary to determine the probability that event e_i follows the sequence of events $e_{i-3}...e_{i-1}$. As shown in Figure 4.3, the probability of executing e_6 in the context of 3 previous events is 1.0 if it follows one of three sequences of events: $< e_1, e_3, e_5 >$, $< e_4, e_7, e_2 >$, or $< e_5, e_4, e_3 >$. In examining the group of sequences given as input to the model, it can be seen that the *only* event which follows the three sequences given here is e_6; therefore, if one of these sequences is seen, the next event must be e_6. Generating test cases from this model uses the highest probabilities given at each node and generates a test case. Therefore, one would expect to see a test case containing each of the sequences given above, followed by e_6, *i.e.*, $< e_1, e_3, e_5, e_6 >$, $< e_4, e_7, e_2, e_6 >$, and $< e_5, e_4, e_3, e_6 >$.

The algorithm shown in Listings 4.1-4.4 generates test cases by constructing and traversing a probabilistic EFG using the method outlined in the previous section.

In the pseudocode shown, the input set of event sequences and the output set of test cases are stored as matrices in which each ith row holds an event sequence and the jth column of a row holds the jth event in the sequence. The algorithm takes two parameters: `EventSeq`, the set of existing test cases and usage profiles, and `history`, the number of previous events on which the probability calculations are to be conditioned. The `history` parameter determines the maximum subsequence length to be used as the conditional probabilities are computed. The final output is `TestSuite`, a set of test cases.

In Step 1, shown in Listing 4.1, each input event sequence is parsed, and subsequences of length `history` are saved. If the value chosen for `history` is larger than the length of the longest input sequence, the subsequences will include the whole input event sequence. As each subsequence is saved, the event following the subsequences is stored and a counter representing $follows$(event, subsequence) is incremented.

In Step 2, shown in Listing 4.2, the probability of each observed *follows* relationship is computed, and stored in a distributions table. For each (subsequence, event) pair, the number of times that subsequence is followed by that event is divided by the number of unique events that follow the subsequence.

In Step 3, shown in Listing 4.3, a sequence of events from the initial event in the application ($INIT$) to each event that appears first in each subsequence is stored. These sequences are used during test case generation to ensure that each test case is a legal sequence of events.

Finally, in Step 4, `TestSuite` is constructed by adding a legal test case (*i.e.*,

Listing 4.1: Store event subsequences from event sequences

```
Input: Event Sequences, history

Step 1: Get all event subsequences of length history
# initialize a set of counters to keep track of occurrences of events
for each event in (input_file) {
  counter[event] = 0;
}

for each event_sequence in ( input_file) {
  for each event in ( event_sequence ) {
    increment counter[event];
   }
   subseq_length = history
   for first_event ( 0 .. length(event_sequence)-subseq_length ) {
     last_event = first_event + subseq_length;
     # get subsequence of events and store in prefixes
     subsequence = event_sequence[ first_event .. last_event];
     add subsequence to prefixes;
     # get the event following this subsequence
     followers{subsequence} = event_sequence[last_event + 1];
     increment follow_count{subsequence}{event};
   }
}
```

Listing 4.2: Compute conditional probabilities

```
Step 2: Get distributions
# Initialize distributions chart for subsequences and events
for each subsequence in ( @prefixes ) {
  for each event in ( @counter ) {
    distributions{event}{subsequence} = 0;
  }
}

# store probability of every subsequence followed by every
# event in distributions
for each event in ( @counter ) {
  for each subsequence in ( @prefixes ) {
    distributions{subsequence}{event} =
         follow_count{subsequence}{event} / num followers{subsequence};
  }
}
```

Listing 4.3: Get path from INIT to first event

```
Step 3: Build sequence of events from INIT..first event in each subsequence

for each event_sequence in (subsequence) {
  for each event in ( subsequence ) {
    initPre{event} = subsequence[ 0 .. event-1 ];
  }
}
```

Listing 4.4: Generate Test Cases

```
Step 4: Generate Test Cases

# save highest probability {event, subsequence} combination
# for each subsequence
# if there's a tie, save all
for each subsequence in ( all subsequences ) {
  for each event in ( all events ) {
    max_prob = 0;
    if (distributions{subsequence}{event} >= max_prob) {
      add distributions{subsequence}{event} to maxVals{subsequence};
    }
  }
}

# maxVals: probability for each subsequence; use in test case creation
# match with DISTRIBS to get the following event and build test cases
for each subsequence in ( distributions ) {
  for each probability in ( maxVals{subsequence} ) {
    test = subsequence . ' ' . event;
    init_test = initPre{subsequence} . ' ' . test;
  }
}

OUTPUT: Set of unique test cases
```

one that begins with `INIT`) for each column maximum in `Distributions`. First, each subsequence is matched with the event that has the greatest probability of following that subsequence, based on the probabilities calculated in Step 2. These combinations of (subsequence, event) are stored in `maxVals`, along with their probability. Test cases are then generated by looping through the `maxVals` data structure and printing test cases composed of the (subsequence, event) sequence of events related to the probability stored in `maxVals`. Before adding the new test case to the output set, a redundancy check is performed against test cases already in the output set, and redundant test cases are not added. Applying the four steps to the 13 example sequences using $H = 1, 2, 3$ yielded the test suites shown in Table 4.1.

Next, these consolidated suites and the reduced suites that were presented in

$INIT, e_1, e_4, e_3, FINAL$	$INIT, e_1, e_3, e_5 FINAL$	$INIT, e_1, e_3, e_5, e_6, FINAL$
$INIT, e_1, e_6, FINAL$	$INIT, e_1, e_4, e_3, e_6, FINAL$	$INIT, e_1, e_4, e_3, e_6, FINAL$
$INIT, e_2, e_3, FINAL$	$INIT, e_1, e_4, e_7, e_2, FINAL$	$INIT, e_1, e_4, e_7, e_2, e_6, FINAL$
$INIT, e_3, e_5, e_6, FINAL$	$INIT, e_1, e_6, FINAL$	$INIT, e_1, e_6, FINAL$
$INIT, e_7, e_2, FINAL$	$INIT, e_2, e_3, e_5, FINAL$	$INIT, e_2, e_3, e_5, e_6, FINAL$
$INIT, e_1, e_4, e_7, FINAL$	$INIT, e_2, e_4, e_7, FINAL$	$INIT, e_2, e_4, e_7, e_2, FINAL$
	$INIT, e_2, e_6, FINAL$	$INIT, e_2, e_6, FINAL$
	$INIT, e_3, e_5, e_4, e_3, FINAL$	$INIT, e_3, e_5, e_4, e_3, e_6, FINAL$
	$INIT, e_3, e_5, e_6, FINAL$	$INIT, e_3, e_5, e_6, FINAL$
	$INIT, e_3, e_6, FINAL$	$INIT, e_7, e_1, e_6, FINAL$
	$INIT, e_7, e_1, e_6, FINAL$	$INIT, e_7, e_2, e_3, e_5, FINAL$
	$INIT, e_7, e_2, e_3, FINAL$	$INIT, e_7, e_2, e_6, FINAL$
	$INIT, e_7, e_2, e_6, FINAL$	
(a) H=1	(b) H=2	(c) H=3

Table 4.1: Example test cases produced from model

Chapter 1 are informally studied. First, the code coverage of these suites is examined. Figure 4.4 illustrates the event handler code used in the `Radio Button Demo` application. The checkboxes shown in Figure 4.4 indicate which test suite executes each line of code. From left to right, the first box represents the original suite, the second box represents the line coverage reduced suite, the third box represents the branch coverage reduced suite, the fourth box represents the method coverage reduced suite, the fifth represents event pair reduced suite and the sixth box represents the suite generated by this model from setting history to 1, 2, and 3. From these checkboxes, it can be seen that the test suites all have more or less the same line coverage.

```
1   RBExample::CircleAction(ActionEvent evt){
2   ⌀⌀⌀⌀⌀⌀currentShape = SHAPE_CIRCLE;
3   ⌀⌀⌀⌀⌀⌀if(created) {
4   □□□□□□ imagePanel.setShape(currentShape);
5   □□□□□□ imagePanel.repaint()}}
```

e_1's Event Handler

```
1   RBExample::SquareAction(ActionEvent evt){
2   ⌀⌀⌀⌀⌀⌀currentShape = SHAPE_SQUARE;
3   ⌀⌀⌀⌀⌀⌀if(created) {
4   □□□□□□ imagePanel.setShape(currentShape);
5   □□□□□□ imagePanel.repaint();}}
```

e_2's Event Handler

```
1   RBExample::ColorAction(ActionEvent evt){
2   ⌀⌀⌀⌀⌀⌀colorText.setEditable(true);
3   ⌀⌀⌀⌀⌀⌀color.setSelected(true);
4   ⌀⌀⌀⌀⌀⌀if (created) {
5   □□□□□□ currentColor = getColor();
6   □□□□□□ imagePanel.setFillColor(currentColor);
7   □□□□□□ imagePanel.repaint();}}
```

e_3's Event Handler

```
1   RBExample::NoneAction(ActionEvent evt){
2   ⌀⌀⌀⌀⌀⌀colorText.setEditable(false);
3   ⌀⌀⌀⌀⌀⌀currentColor = COLOR_NONE;
4   ⌀⌀⌀⌀⌀⌀if (created) {
5   □□□□□□ imagePanel.setFillColor(currentColor);
6   □□□□□□ imagePanel.repaint();}}
```

e_4's Event Handler

```
1   RBExample::CreateAction(ActionEvent evt) {
2   ⌀⌀⌀⌀⌀⌀currentColor = getColor();
3   ⌀⌀⌀⌀⌀⌀imagePanel.setFillColor(currentColor);
4   ⌀⌀⌀⌀⌀⌀imagePanel.setShape(currentShape);
5   ⌀⌀⌀⌀⌀⌀imagePanel.repaint();
6   ⌀⌀⌀⌀⌀⌀created = true;}
```

e_6's Event Handler

```
1   RBExample::ResetAction(ActionEvent evt){
2   ⌀⌀⌀⌀⌀⌀square.setSelected(true);
3   ⌀⌀⌀⌀⌀⌀none.setSelected(true);
4   ⌀⌀⌀⌀⌀⌀colorText.setText("black");
5   ⌀⌀⌀⌀⌀⌀colorText.setEditable(false);
6   ⌀⌀⌀⌀⌀⌀currentShape = SHAPE_NONE;
7   ⌀⌀⌀⌀⌀⌀imagePanel.setShape(currentShape);
8   ⌀⌀⌀⌀⌀⌀currentColor = COLOR_NONE;
9   ⌀⌀⌀⌀⌀⌀imagePanel.setFillColor(currentColor);
10  ⌀⌀⌀⌀⌀⌀imagePanel.repaint();}
```

e_7's Event Handler

```
1   ImagePanel::paintComponent(Graphics g) {
2   ⌀⌀⌀⌀⌀⌀clear(g);
3   ⌀⌀⌀⌀⌀⌀Graphics2D g2d = (Graphics2D)g;
4   ⌀⌀⌀⌀⌀⌀if (currentShape == SHAPE_CIRCLE) {
5   ⌀⌀⌀⌀⌀⌀ if (currentColor == COLOR_NONE) {
6   ⌀⌀⌀□⌀⌀ g2d.setPaint(Color.black);
7   ⌀⌀⌀□⌀⌀ g2d.draw(circle);}
8   ⌀⌀⌀⌀⌀⌀ else {
9   ⌀⌀□⌀⌀⌀ g2d.setPaint(currentColor);
10  ⌀⌀□⌀⌀⌀ g2d.fill(circle);}}
11  ⌀⌀⌀⌀⌀⌀else if (currentShape == SHAPE_SQUARE) {
12  ⌀⌀⌀⌀⌀⌀ if (currentColor == COLOR_NONE) {
13  ⌀⌀⌀□⌀⌀ g2d.setPaint(Color.black);
14  ⌀⌀⌀□⌀⌀ g2d.draw(square);}
15  ⌀⌀⌀⌀□⌀ else {
16  ⌀⌀⌀⌀□⌀ g2d.setPaint(currentColor);
17  ⌀⌀⌀⌀□⌀ g2d.fill(square);}}}
18    ImagePanel:setFillColor(int inputColor) {
19  ⌀⌀⌀⌀⌀⌀switch(inputColor) {
20  ⌀⌀⌀⌀⌀⌀ case COLOR_BLACK:
21  ⌀⌀□□⌀⌀ currentColor=Color.black;
22  ⌀⌀□□⌀⌀ break;
23  ⌀⌀⌀⌀⌀⌀ case COLOR_RED:
24  ⌀⌀□⌀⌀⌀ currentColor=Color.red;
25  ⌀⌀□⌀⌀⌀ break;
26  ⌀⌀⌀⌀⌀⌀ case COLOR_GREEN:
27  ⌀⌀⌀□⌀⌀ currentColor=Color.green;
28  ⌀⌀⌀□⌀⌀ break;
29  ⌀⌀⌀⌀⌀⌀ default:
30  ⌀⌀⌀⌀⌀⌀ currentColor=Color.gray;}}
31    String ImagePanel::getColor() {
32  ⌀⌀⌀⌀⌀⌀if (color.isSelected()){
33  ⌀⌀⌀⌀⌀⌀ return colorText.getText();}
34  ⌀⌀⌀⌀□⌀else {
35  ⌀⌀⌀□□⌀ return colorText.setText("gray");}
```

The ImagePanel Class

Figure 4.4: Some Source Code for the Radio Button GUI Example.

Upon close inspection, it can be seen there is a fault in this code. When line 33 of the `ImagePanel::getColor()` method is executed, it causes the `ImagePanel` to crash. This is because `getText()` expects to find a value in the text box (widget w_5). If there is no value, `getText()` will return NULL, which is not properly handled in this code, causing an uncaught exception to be thrown.

This fault is not detected by the original suites or the reduced suites; it is only detected by the test case $< e_1, e_4, e_3, e_6 >$ generated by the CONTEST model for both $H = 2$ and $H = 3$. Referring back to Figure 4.2, the table related to node e_6 shows the probability of the sequence $< e_4, e_3, e_6 >$ is 1.0. Likewise, Figure 4.3 shows that the probability of the sequence $< e_4, e_3, e_6, FINAL >$ is 1.0. The algorithm will choose these sequences to create test cases, and prepend events to reach the first event in the sequence, in this case, e_4. The value of test suite consolidation is seen even in this small example.

4.3 Empirical Study

To evaluate the CONTEST algorithm, an empirical study comparing test suites generated by the CONTEST algorithm to existing test suites on the basis of three dimensions: *fault detection*, *cost of testing*, and *code coverage* was conducted. Using the CONTEST algorithm, five test suites were generated for the study by varying the `history` parameter from one to five ($T_{H1}, T_{H2}, ..T_{H5}$). Three reduced suites, reduced by line coverage (T_{line}), method coverage (T_{method}) and event pair coverage (T_{pair}), were used as controls in the study. This gives a total of nine

suites, including the original suite (T_{orig}), to be considered. There is no interaction between the test cases, therefore the results of one does not influence the results of another.

The goal of this study is to *compare the effectiveness of a test suite composed of existing test cases and usage profiles treated as test cases (T_{orig}) to a suite of test cases generated by the CONTEST algorithm.* Restating this goal using the Goal Question Metric (GQM) Paradigm [2], the goal for this research is as follows:

> Analyze the **consolidated test suites**
> for the purpose of **determining effectiveness**
> with respect to **existing test suites**
> from the point of view of the **tester/researcher**
> in the context of **event driven systems**.

More specifically, this study enabled analysis of the CONTEST technique by answering the following questions:

RQ4.1 Which suite is the most effective at fault detection?

RQ4.2 Which suite has the lowest cost of testing?

RQ4.3 Which suite has the best code coverage?

Using T_{orig} to populate the probabilistic model, the first question is asking which test suite is the more effective at finding faults. Fault detection was computed by running each test case on each subject application and recording whether or not the application crashed, or failed, during test case execution. After collecting the set of failures, each was manually linked to the fault that caused

it. For each suite, the number of defects detected was compared by computing the total number of faults detected by each *suite* or $F(suite)$, where *suite* $\in \{T_{orig}, T_{H1}, T_{H2}...T_{H5}, T_{line}, T_{method}, T_{pair}\}$.

The second question relates to the cost of generating and executing test cases, and is aimed at determining which suite will be cheaper to execute. In this study, cost, or $c(suite)$ is measured in the number of test cases, for *suite* $\in \{T_{orig}, T_{H1}, T_{H2}...T_{H5}, T_{line}, T_{method}, T_{pair}\}$. In the context of this study, calculating cost as the number of test cases in the suite is reasonable since the overhead required to run each test case, including starting the test case execution framework, starting the application, and ending the application and framework, consumed more computation time than any other activity. The difficulty in choosing a consistent cost measure is acknowledged as a threat to validity, as noted in Section 4.3.5.

The third question addresses the difference in code coverage of the original test suite (T_{orig}) as compared to the code coverage of the other generated test suites. Code coverage, $coverage(suite)$ for *suite* $\in \{T_{orig}, T_{H1}, T_{H2}...T_{H5}, T_{line}, T_{method}, T_{pair}\}$ is computed in terms of line, method, and block.

By answering the questions posed in this study, the most effective value of `history` for each application will also be revealed. The intuition behind choosing the value for the history parameter is that there is a point where the additional computation required by choosing a larger value for the history parameter is balanced by an increase in the fault detection of the suite.

4.3.1 Subject Applications

Four popular, open source Java applications were chosen and downloaded from SourceForge for this study:

1. **CrosswordSage 0.3.5**[2], a popular tool for creating and solving professional-looking crossword puzzles with built-in word suggestion capabilities, with an all-time activity rate of 78.28%.

2. **FreeMind 0.8.0**[3], a very popular mind-mapping application, with an all-time activity rate of 100%.

3. **GanttProject 2.0.1**[4], a project scheduling application featuring Gantt chart, resource management, calendars, and the option to import/export MS Project, HTML, PDF, and spreadsheets, with an all-time activity rate of 99.98%.

4. **jMSN 0.9.9b2**[5], a clone of MSN Messenger, including instant messaging, file sharing, and additional chat features standard in MSN Messenger, with an all-time activity rate of 98.62%.

These applications were chosen for several reasons. All of the applications have an active developer community and high all-time-activity scores on SourceForge, with three of the applications above 90%. CrosswordSage was chosen partially because it is fairly new (first released in 2005) and yet has an activity score of

[2]http://sourceforge.net/projects/crosswordsage
[3]http://sourceforge.net/projects/freemind
[4]http://sourceforge.net/projects/ganttproject
[5]http://sourceforge.net/projects/jmsn

almost 80%. Finally, these applications have been released in several versions and have undergone quality assurance prior to each release.

4.3.2 Tools

This study was supported by several pre-existing tools and a new test case generation tool (can be downloaded at http://guitar.sourceforge.net) created from the algorithm described in Section 4.2.

The **GUI Testing FrAmewoRk (GUITAR)** was used to perform this study [39]. The **JavaGUIRipper**, one of the tools in the GUITAR suite, was used to glean the structure of the subject applications. By using Java Reflection, the JavaGUIRipper creates an XML file that represents the windows, menu items, and buttons present in the GUI, including the actions that are executed when those items are selected.

Usage profiles can be captured by a tool in GUITAR's family of applications called the **Profiler** [41]. (The Profiler does not currently belong to GUITAR's canonical, publicly available set of tools.) Running the subject application through Java Reflection, the Profiler attaches its own event handlers to each JButton, JTextArea, and JMenuItem that becomes visible. When one of the Profiler's event handlers is triggered, the Profiler records an identifier for the widget and the type of event.

Test cases can be created using a **parameterized test case generator**, developed in previous work [66]. Test cases are generated to exhaustively cover

events up to n in the EFG, where n is given as a parameter to the generator.

The **N-Gram model test case generator**, based on the model presented in Section 4.1 and the CONTEST algorithm described in Section 4.2, takes event sequence-based test cases, such as those from the Profiler and the parameterized test case generator, as input to build consolidated test suites. The model is generated from the input sequences and probabilities are assigned based on sequences of events observed in the input suites. A new test suite is output from this model, representing a consolidation of the input suites.

Another tool in the GUITAR tool suite, the **JavaGUIReplayer**, was used for test case execution. The JavaGUIReplayer is a framework that opens the application under test and replays XML test cases containing details on the steps to be performed. Each event is executed on the GUI, and the state of the GUI is recorded after each step. The state is saved in XML files that can be examined to determine which test cases failed and why. The JavaGUIReplayer also prepends events to the front of the test case, as necessary, to ensure that the first event in the test case is available from the start state of the application.

In preparation for this study, a database for text-field values was created. To automate test case replaying, a database that contains one instance for each of the text types in the set {*negative number, real number, long file name, empty string, special characters, zero, existing file name, non-existent file name*} was used. Note that if a text field is encountered in the GUI (represented as an event called `type-in-text`), one instance for each text type is tried in succession. The test oracle used for this study was developed to detect crashes for these applications.This

approach has been used before and has been found to be useful [61, 66, 5].

4.3.3 Implementation

The study presented in this chapter relied upon particular representations in order to tie the process together.

4.3.3.1 Representations, Notations, and Examples

Usage profiles and test cases are both stored in an XML format understood by the JavaGUIReplayer. Examples of the files relevant to this process, to include a usage profile, a mapping file, the original suites coded file, and a resulting test case, are shown in Figures 4.5 and 4.8. For the sake of space, only a few attributes of each GUI widget are shown, and only for the first step. In a full profile or test case, there are 16 attributes for each step.

Figure 4.5 shows a partial usage profile for GanttProject. This profile is composed of six steps (each step is surrounded by `<Step>` and `</Step>` tags), which together create a new project in GanttProject, and set some of the project specifics in the "Create new project" window. Finally, Cancel is selected, which will cancel all of the user actions for creating the new project. The test cases generated using the parameterized test case generator will also follow this format.

Figure 4.6 is an example of the file used by the test case generation algorithm. Each line in this file represents a usage profile or an existing test case in an abbreviated format; each number represents one event in the profile or test case.

```
<Profile>
    <Step>
       <Window>GanttProject_1</Window>
       <Component>New..._R_33</Component>
       <Action>doClick</Action>
       <WindowFlag>FALSE</WindowFlag>
       <Attributes>
           <Property>
             <Name>IconImage</Name>
             <Value>IconImage_24</Value>
           </Property>
           <Property>
             <Name>Type</Name>
             <Value>RESTRICTED</Value>
           </Property>
           <Property>
             <Name>ReplayableAction</Name>
             <Value>doClick</Value>
           </Property>
           <Property>
             <Name>Visible</Name>
             <Value>TRUE</Value>
           </Property>
           <Property>
             <Name>Enabled</Name>
             <Value>TRUE</Value>
           </Property>
       </Attributes>
    </Step>
```

```
    <Step>
       <Window>Create new project_2</Window>
       <Component>AutoText_R_0</Component>
       <Action>setText_String_5</Action>
       <WindowFlag>FALSE</WindowFlag>
    </Step>
    <Step>
       <Window>Create new project_2</Window>
       <Component>AutoText_R_2</Component>
       <Action>setText_String_5</Action>
       <WindowFlag>FALSE</WindowFlag>
    </Step>
    <Step>
       <Window>Create new project_2</Window>
       <Component>AutoText_R_1</Component>
       <Action>setText_String_5</Action>
       <WindowFlag>FALSE</WindowFlag>
    </Step>
    <Step>
       <Window>Create new project_2</Window>
       <Component>AutoText_R_9</Component>
       <Action>setText_String_5</Action>
       <WindowFlag>FALSE</WindowFlag>
    </Step>
    <Step>
       <Window>Create new project_2</Window>
       <Component>Cancel_R_23</Component>
       <Action>doClick</Action>
       <WindowFlag>FALSE</WindowFlag>
    </Step>
</Profile>
```

Figure 4.5: Partial usage profile for GanttProject

```
1 2 3 4 7 9
1 2 10 3 12 7
1 2 3 12 5 7
```

Figure 4.6: An example coded file used to represent event sequences

The mapping file, shown in Figure 4.7, is the translator between the input event sequences, *i.e.*, usage profile or test case, and the test case generation input file. The portion of the map file shown in Figure 4.7 describes the events in the sample file and ties to the first line of the input file.

```
1 <Window>GanttProject_1</Window><Component>New..._R_33</Component>
2 <Window>Create new project_2</Window><Component>AutoText_R_0</Component>
3 <Window>Create new project_2</Window><Component>AutoText_R_1</Component>
4 <Window>Create new project_2</Window><Component>AutoText_R_9</Component>
7 <Window>Create new project_2</Window><Component>Cancel_R_23</Component>
9 <Window>Create new project_2</Window><Component>AutoText_R_2</Component>
```

Figure 4.7: An example map file used to link the coded file to executable events

The output of the **N-Gram model test case generator** is a multi-line file of integer sequences representing the generated test suite. The resulting file was expanded into test case events, based on the numbers assigned in the mapping, and test cases were generated. An example of a test case generated by the N-Gram model test case generator, in the integer sequence format, is: `1 2 9 4 12 3 1`. This exact sequence of events did not occur in the input set of event sequences. Figure 4.8 shows this generated test case in executable form, after the integers are mapped back to the events they represent.

```
<Testcase>
   <Step>
      <Action>doClick</Action>
      <WindowFlag>FALSE</WindowFlag>
      <Window>GanttProject_1</Window>
      <Component>New..._R_33</Component>
    </Step>
    <Step>
       <Action>setText_String_5</Action>
       <WindowFlag>FALSE</WindowFlag>
       <Window>Create new project_2</Window>
       <Component>AutoText_R_0</Component>
    </Step>
    <Step>
       <Action>setText_String_5</Action>
       <WindowFlag>FALSE</WindowFlag>
       <Window>Create new project_2</Window>
       <Component>AutoText_R_2</Component>
    </Step>
    <Step>
       <Action>setText_String_5</Action>
       <WindowFlag>FALSE</WindowFlag>
       <Window>Create new project_2</Window>
       <Component>AutoText_R_9</Component>
    </Step>
```

```
    <Step>
       <Action>setText_String_5</Action>
       <WindowFlag>FALSE</WindowFlag>
       <Window>Create new project_2</Window>
       <Component>AutoText_R_4</Component>
    </Step>
    <Step>
       <Action>setText_String_5</Action>
       <WindowFlag>FALSE</WindowFlag>
       <Window>Create new project_2</Window>
       <Component>AutoText_R_1</Component>
    </Step>
    <Step>
       <Action>doClick</Action>
       <WindowFlag>FALSE</WindowFlag>
       <Window>GanttProject_1</Window>
       <Component>New..._R_33</Component>
    </Step>




</Testcase>
```

Figure 4.8: Portion of an automatically generated test case

4.3.4 Procedure

To answer the questions posed in this study, several steps were executed. Figure 4.9 gives a graphical representation of the steps described here. First, user pro-

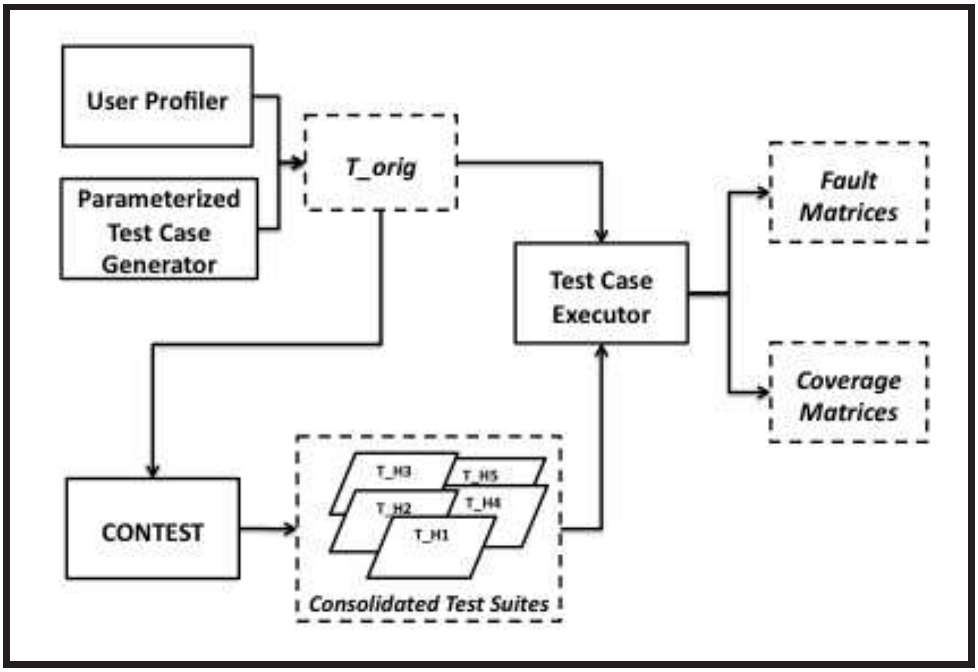


Figure 4.9: Empirical Study Procedure

files were collected from 15 students who participated in a monitored, task-based exercise using the User Profiler tool that is part of the GUITAR suite, and described in Section 4.3.3. The students were given a list of tasks to complete on each application, but were not told which steps to use to create the final product. Because there are many choices to reach the same end result, this method provided different usage profiles for each person, while still ensuring there would be some similarity in the parts of the application exercised.

These profiles, stored as sequences of events in XML formatted files, were then converted to the proper format to be treated as test cases by the JavaGUIReplayer, also described in Section 4.3.3. This process took approximately 30 minutes to 1 hour for each set of profiles. Executing the user profile-based test cases took approximately 10 to 12 hours for each suite. Table 4.2 shows the number of profiles

collected for each application. CrosswordSage and jMSN had under 500, while GanttProject had nearly 1,300 and FreeMind had approximately 6,500.

Next, test suites were created using the parameterized test case generator described in Section 4.3.3. Several suites were generated for each subject application; each suite focused on one functional area, *e.g.*, separate test suites were created to test opening a file, editing a file and formatting a file. The number of functional areas varied for each application, ranging from 8 in CrosswordSage to 18 in GanttProject. In some cases, the functionality is associated with the GUI itself, such as `autocombo`, `autogen` or `autotext`, indicating the type of GUI functionality in that portion of the application. For example, `autocombo` represents a combo box, `autogen` usually represents clickable buttons in the GUI, and `autotext` represents a text field that the user will fill in. The rest of the suites in the parameterized test cases represent functionality from a user standpoint, such as `new`, creating a new project in GanttProject, and `logonoff`, logging in or out on jMSN. A total of 1,426 test cases were generated for CrosswordSage, 44,856 test cases for FreeMind, 27,835 test cases for GanttProject, and 4,242 test cases for jMSN. Together, the profiles and parameterized test case suites are called T_{orig}. The names and sizes of all of the original suites are shown in Table 4.2.

Next, each test case in T_{orig} was run using a test case executor, in this case the JavaGUIReplayer, spread across a cluster of PCs running Linux, using from 4-10 machines at a time, as they were available. The execution of the test suites generated logs that provide fault and coverage information collected during execution. From these logs, fault and coverage matrices were created for each suite, detailing fault

Suite	CrosswordSage		FreeMind		GanttProject		jMSN	
Profiles		427		6460		1296		392
Parameterized Test Cases	action	24	autocombo	678	autogen	2937	autogen	69
	autogen	248	autogen	14238	edit	690	file	138
	autotext	124	autotext	226	export	1260	help	991
	edit	27	browse	226	help	356	langloc	111
	file	793	edit	3390	humanprop	89	logonoff	125
	find	62	file	4722	import	1406	options	600
	help	24	format	7850	new	2862	status	621
	tools	124	help	454	newresource	89	tools	69
			insert	7742	newtask	89	userlist	690
			maps	452	open	6	view	828
			mindmap	226	pert	89		
			navigate	2260	print	89		
			tools	1262	project	1908		
			view	1130	resources	178		
					save	1893		
					task	3		
					taskprop	13339		
					view	552		

Table 4.2: Size of original test suites (T_{orig}) for each application

detection and coverage per test case.

The generated test cases were also used as input to CONTEST. First, each test case was distilled into a sequence of integers (referred to as `Event Sequences` in Section 4.2) and a mapping from each integer to the textual event identifier it represents was created. A test suite was represented in one file, with one line representing each test case. For instance, one file encoded all of T_{orig}. This pre-processing of T_{orig} took approximately 5 minutes.

Next, the Perl implementation of CONTEST was executed using values of 1 through 5 for `history`, which took under 4 minutes per suite. Each of these values produced a new consolidated test suite, indicated in Figure 4.9 as T_Hn where n is the value of `history` used in that run. Using the mapping from integers to event identifiers, the CONTEST file was expanded into XML test cases in the format

expected by the test case executor. This post-processing to generate test cases took from 3 to 45 minutes, depending on the application. Lastly, the CONTEST suites were executed on the JavaGUIReplayer, producing fault and coverage matrices. Test case execution for the CONTEST suites took 6 to 10 hours.

Finally, as controls to this study, T_{orig} was reduced based on line, method and event pair coverage, using the HGS algorithm discussed in Chapter 1 [17].

4.3.5 Threats to Validity

The results of this study should be interpreted with some deference to threats to validity. First, due to the desire to use the existing GUITAR infrastructure, and to compare these results to those posted by previous graduate student researchers, subject applications developed in Java were used. Therefore, the study gives no information on how the results would translate to other development languages.

Second, although the GUI for each application is different, these subject applications do not reflect all possible classes of GUIs. Third, the majority of the application code is written for the GUI, meaning the results may not be consistent for applications with a simple GUI and complex underlying business logic.

Fourth, although the applications chosen for this study have undergone quality assurance, they are open-source and developed by a team of volunteer developers, leading to the possibility that they are more prone to bugs than professionally-developed software. Fifth, measuring cost by the number of test cases is valid using this infrastructure, however, this measure may not translate in another infrastruc-

ture if the time to execute each test step outweighs the setup and tear-down time.

4.4 Results

This section provides the results of each research question posed in the study as well as the metric(s) used to determine the results.

4.4.1 Fault Detection

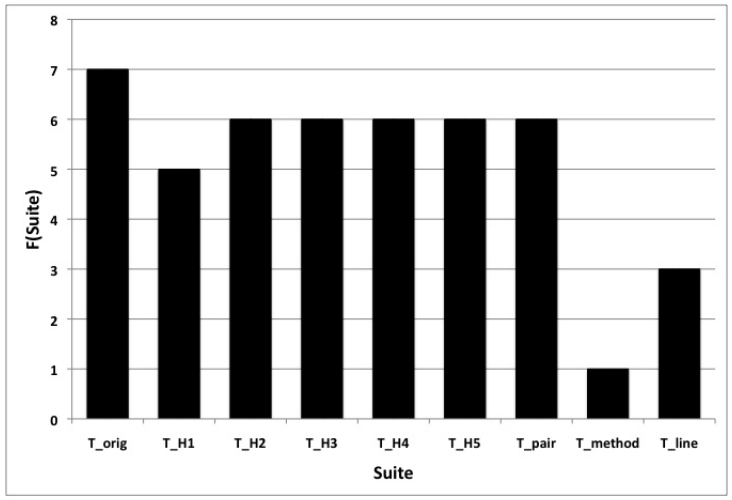
RQ4.1: Which suite is the most effective at fault detection?

Metrics: *Number of faults detected*

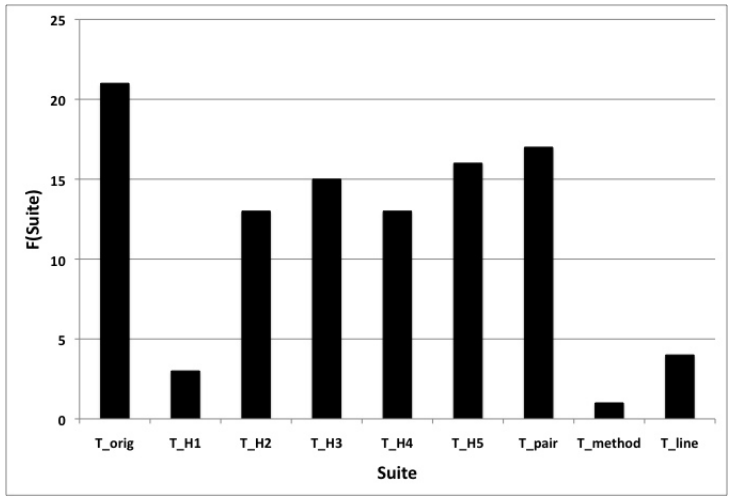
Figure 4.10 shows the fault detection of each application. The method-reduced suites, T_{meth} for each application performed the worst, detecting 0 or 1 fault in each. T_{line} does not do much better, performing the next worse, and tied with T_{H1} in many cases. jMSN's T_{H5} is an outlier in these results and will be further discussed later. The best fault detection results are garnered by the CONTEST suites, with T_{pair} doing well in some cases. The fault detection of CrosswordSage's T_{orig} suite is one more than the CONTEST suites. FreeMind's T_{orig} detected 5 more faults than the best CONTEST suite, T_{H5}. T_{pair} consistently outperformed the other reduction methods, T_{line} and T_{meth}.

Some of these results can be explained by looking at the length of the test cases in T_{orig}. Figure 4.11 shows the number of test cases of each length (generally length 2 through 5) for each application. Combining this information with fault detection helps to explain the $F()$ values. In CrosswordSage, the T_{orig} test suite is

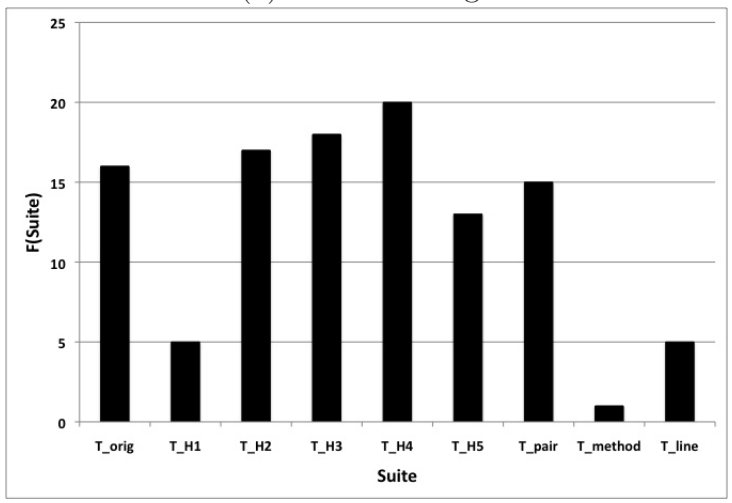
dominated by test cases of length 4. The CONTEST test suites have fewer length 4 test cases than T_{orig} and they also detect less faults. FreeMind's CONTEST suites show a decrease in fault detection between T_{H3} and T_{H4} and an increase from T_{H4} to T_{H5}, which correspond to the decrease in test cases of length 4 and 5, from T_{H3} and T_{H4}. CONTEST suites T_{H3} and T_{H5} have the same number of test cases in length 4 and 5. GanttProject's T_{orig} is dominated by length 3 test cases, as are T_{H2}, T_{H3}, and T_{H4}. T_{H5} doesn't have any length 3 test cases, and there is a significant drop in fault detection as compared to T_{H4}. jMSN's T_{orig} is dominated by length 2 and 3 test cases, as are T_{H2} and T_{H3}. Further, the fault detection for the CONTEST suites increases when length 3 test cases are introduced (in T_{H2}) and decreases when length 2 test cases are lost (in T_{H4}) and again when length 3 test cases are lost (in T_{H5}).



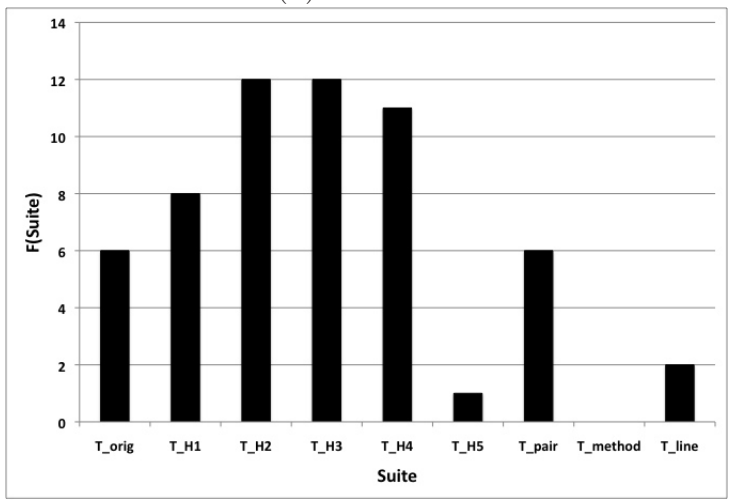
(a) CrosswordSage



(b) FreeMind

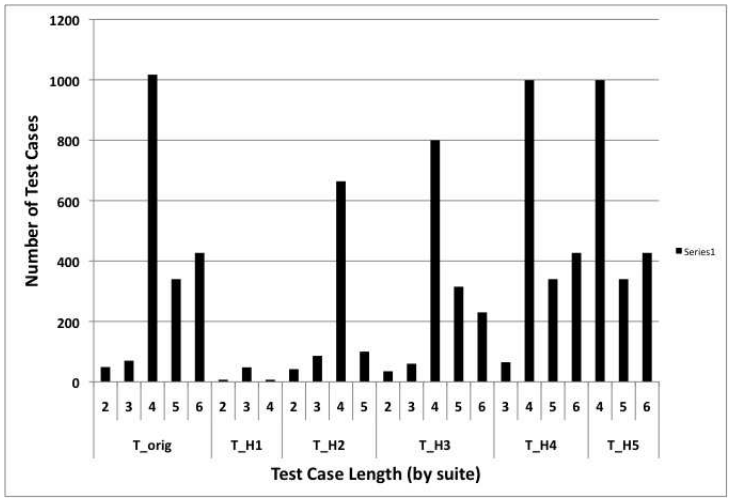


(c) GanttProject

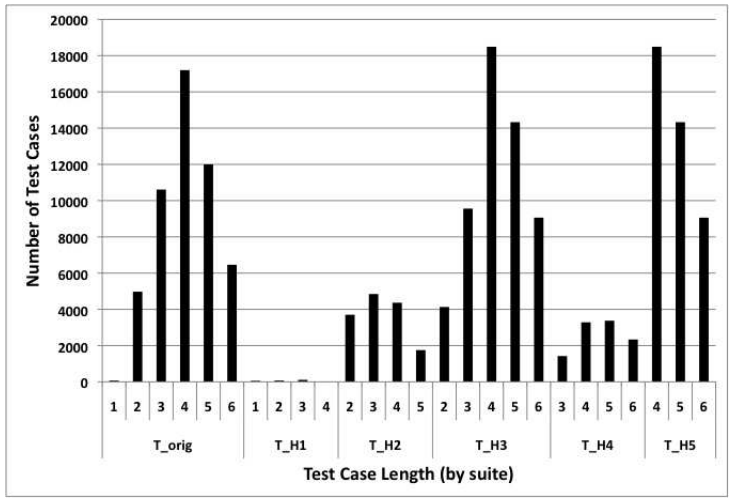


(d) jMSN

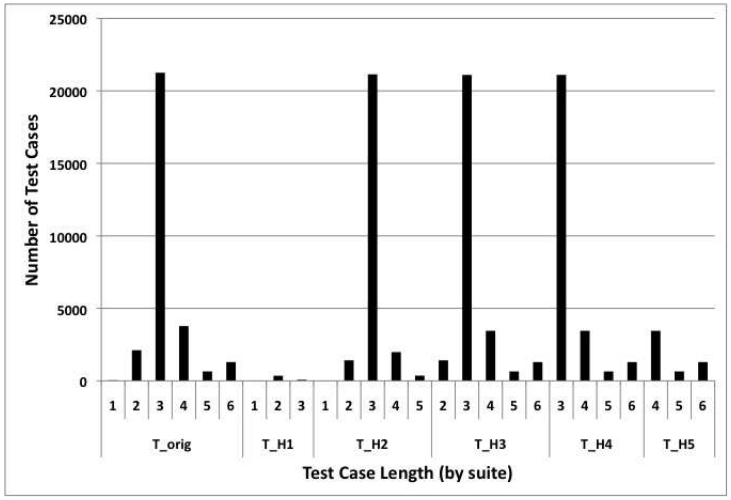
Figure 4.10: Faults detected



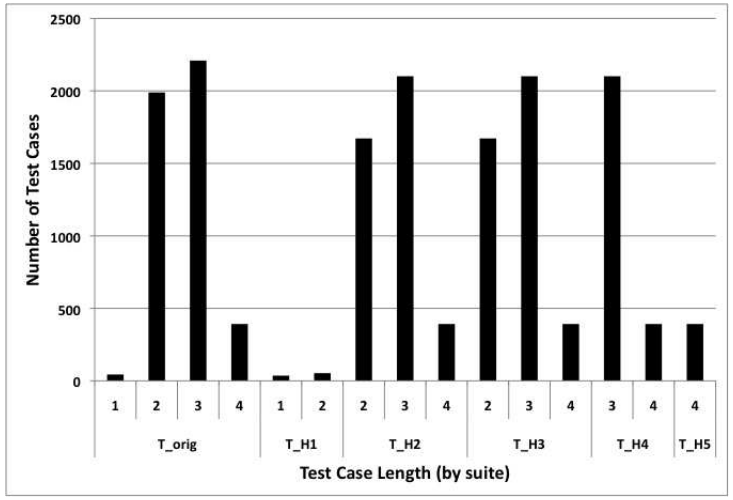
(a) CrosswordSage



(b) FreeMind



(c) GanttProject



(d) jMSN

Figure 4.11: Histograms showing test case length

Application	Suite	Fault Id 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	Total
Crossword	T_{H1}		x	x	x	x		x																					5
Sage	T_{H2}		x	x	x	x	x	x																					6
	T_{H3}	x	x	x	x	x		x																					6
	T_{H4}	x	x	x	x	x		x																					6
	T_{H5}	x	x	x	x	x		x																					6
	T_{orig}	x	x	x	x	x	x	x																					7
	T_{stmt}		x		x		x																						3
	T_{method}		x																										1
	T_{pair}		x	x	x	x	x	x																					6
Free	T_{H1}						x	x				x																	3
Mind	T_{H2}	x	x	x	x		x	x		x	x	x	x	x	x	x													13
	T_{H3}		x	x	x	x	x	x	x	x	x	x	x	x	x	x		x											15
	T_{H4}	x	x		x		x	x		x	x	x	x	x	x	x		x											13
	T_{H5}		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x											16
	T_{orig}		x	x	x	x	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x					21
	T_{stmt}						x					x		x		x													4
	T_{method}									x																			1
	T_{pair}		x	x	x		x	x	x	x	x	x	x	x	x	x		x	x			x		x					17
Gantt	T_{H1}						x		x				x												x	x			5
Project	T_{H2}	x	x			x	x		x		x		x	x	x	x	x		x		x	x		x	x	x			17
	T_{H3}	x	x			x	x	x	x		x	x	x	x	x	x	x		x			x		x	x	x			18
	T_{H4}	x	x	x	x	x	x		x	x		x	x	x	x	x	x	x		x		x	x		x	x			20
	T_{H5}	x	x	x			x		x			x		x	x		x		x			x			x	x			13
	T_{orig}	x		x					x		x	x	x	x	x	x	x	x	x			x		x			x	x	16
	T_{stmt}														x			x				x		x				x	5
	T_{method}								x																				1
	T_{pair}	x		x					x		x	x	x	x	x	x	x		x			x		x			x	x	15
jMSN	T_{H1}	x			x	x	x	x			x	x	x																8
	T_{H2}	x	x	x	x	x	x	x	x	x	x	x	x																12
	T_{H3}	x	x	x	x	x	x	x	x	x	x	x	x																12
	T_{H4}	x	x		x	x	x	x	x	x	x	x	x																11
	T_{H5}												x																1
	T_{orig}				x		x					x		x	x	x													6
	T_{stmt}											x		x															2
	T_{method}																												0
	T_{pair}				x		x					x		x	x	x													6

Figure 4.12: Faults detected in all four subject applications

Figure 4.12 shows the individual faults detected by each suite, for each subject application. It can be seen from this table that as the history value is increased, different faults are found. For example, CrosswordSage's T_{H2} found Fault #6, which was not detected by any of the other CONTEST suites, while three of the suites, T_{H3}, T_{H4}, and T_{H5} found Fault #1. FreeMind's CONTEST suites detected two faults, Fault #1 and Fault #16, that were not detected by any of the other suites. In this case, T_{H2} and T_{H4} had similar results, as did T_{H3} and T_{H5}. T_{H5} detected a new fault, not detected by any of the other suites. GanttProject's CONTEST suites were most effective for `history` values of 2, 3 and 4. T_{H2} found two faults not detected by the other suites, while T_{H3} found one and T_{H4} found four. jMSN's T_{H2} found one fault that was not found by the other suites.

There are also cases where increasing history decreases fault detection, such as jMSN's T_{H5} suite. As discussed above, this is related to the length of the test cases, and more specifically the length of the input suite, T_{orig}.

4.4.2 Cost

RQ4.2: Which suite has the lowest cost of testing?

Metrics: *Number of test cases in suite*

Cost was computed and compared for each test suite, generated with varying levels of history from 1 through 5. Figure 4.13 shows the difference in the cost of each suite. For CrosswordSage, the CONTEST suites are linearly increasing as history is increased, for history 1 - 4; T_{H5} is slightly smaller than T_{H4}. All of the

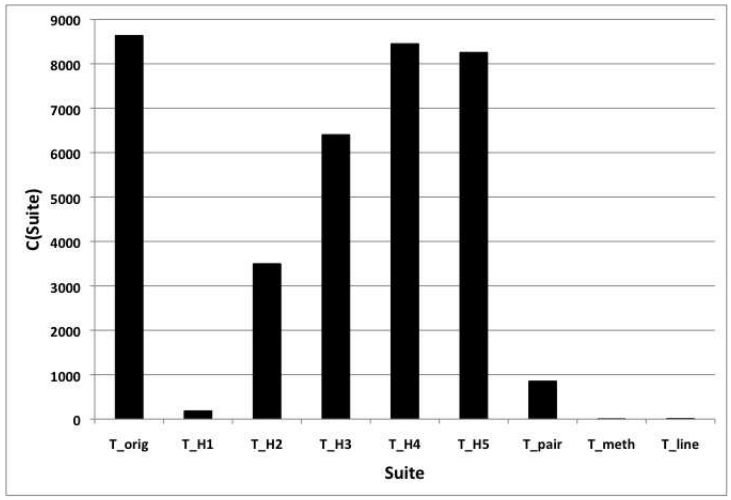
generated suites are smaller than T_{orig}. FreeMind's suites do not follow the same pattern; T_{H3} and T_{H5} are very similar in size to T_{orig} while the others are small. GanttProject has a very small T_{H1} and T_{H5}, while the T_{H2}, T_{H3}, and T_{H4} are closer in size to T_{orig}. jMSN's generated suites decrease in size for histories 2-5; an opposite trend from that witnessed in CrosswordSage. Referring back to Figure 4.11, these suite size trends are related to the length of the input sequences, T_{orig}.

4.4.3 Code Coverage

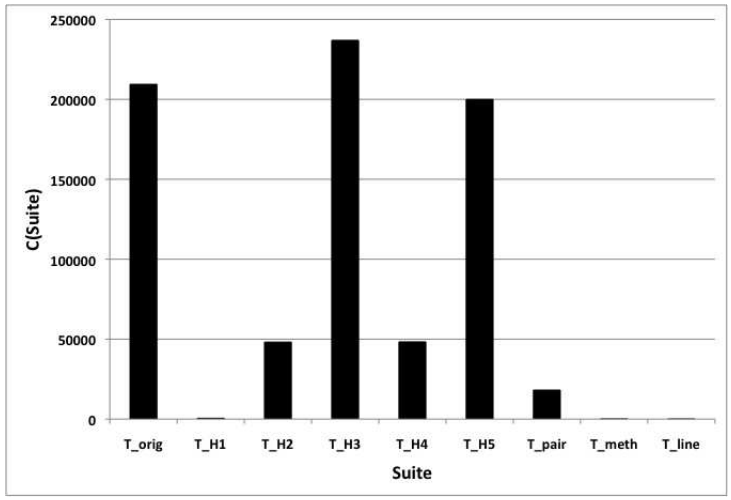
RQ4.3: Which suite has the best code coverage?

Metrics: *Line coverage, Method Coverage*

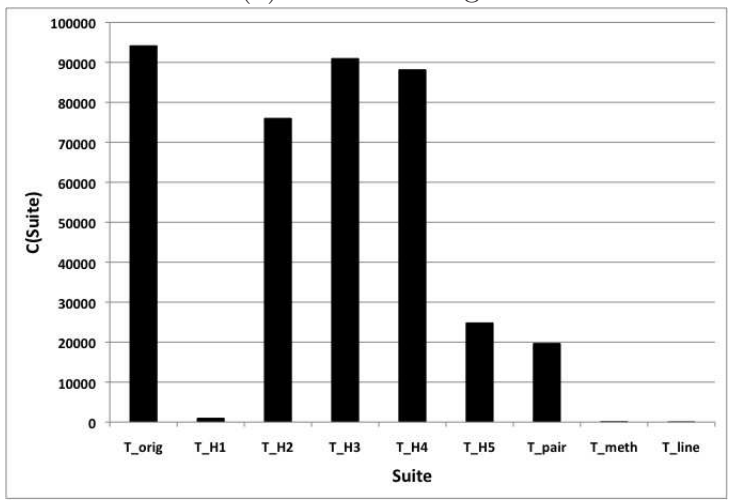
Code coverage was computed and compared across test suites for each application. Figure 4.15 shows the difference between the coverage in the original, reduced and all five CONTEST-generated test suites. For each application, coverage was similar across the test suites, however the CONTEST suites were slightly better in each case.



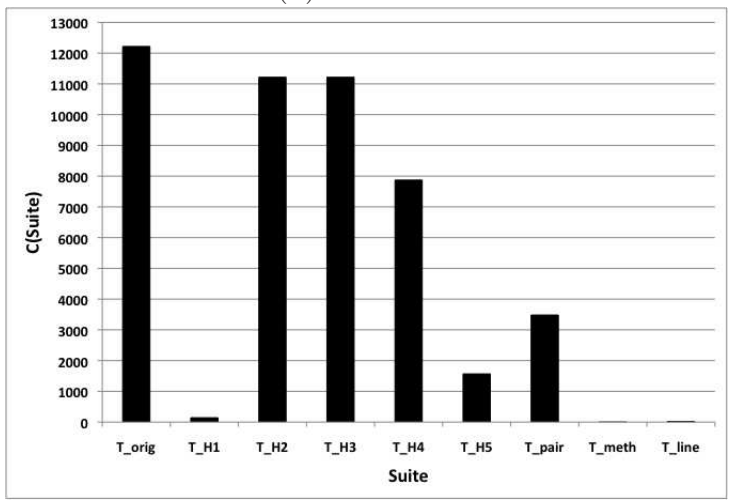
(a) CrosswordSage



(b) FreeMind

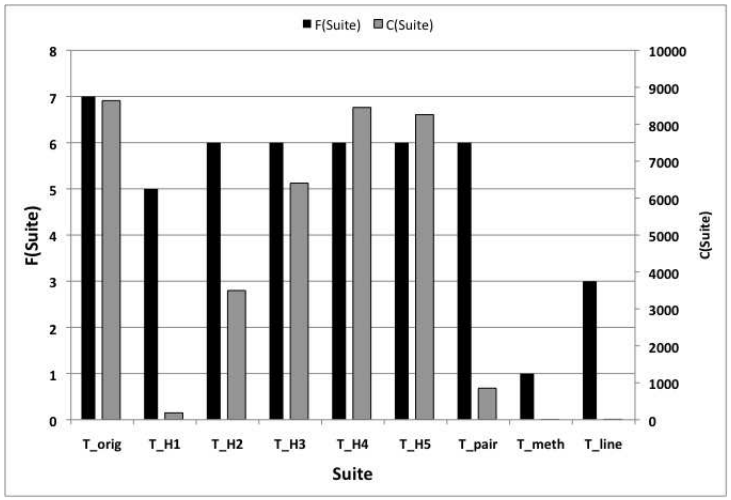


(c) GanttProject

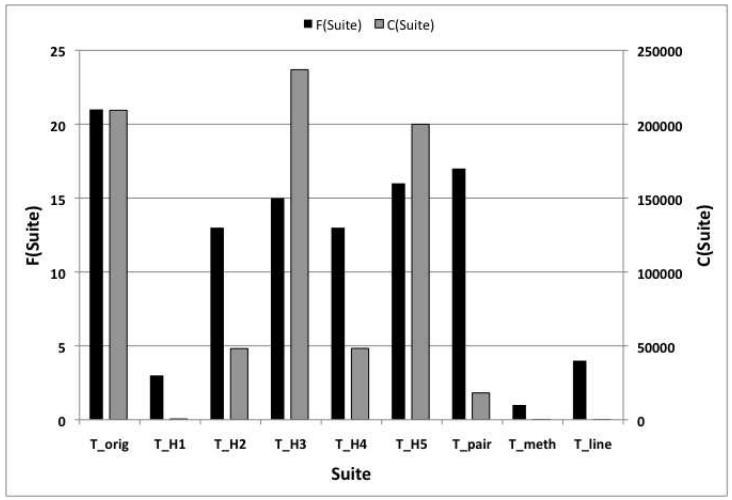


(d) jMSN

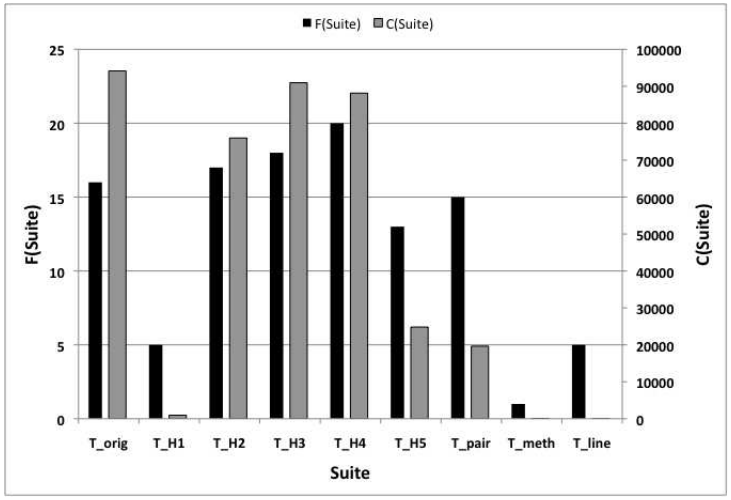
Figure 4.13: Cost for each subject application's test suites



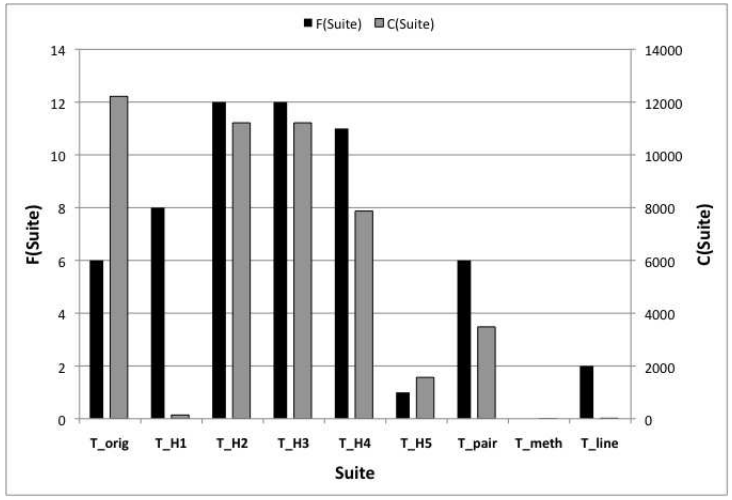
(a) CrosswordSage



(b) FreeMind

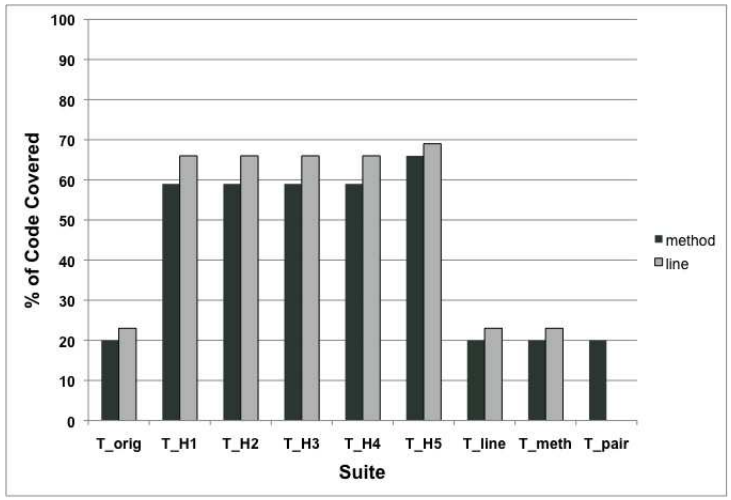


(c) GanttProject

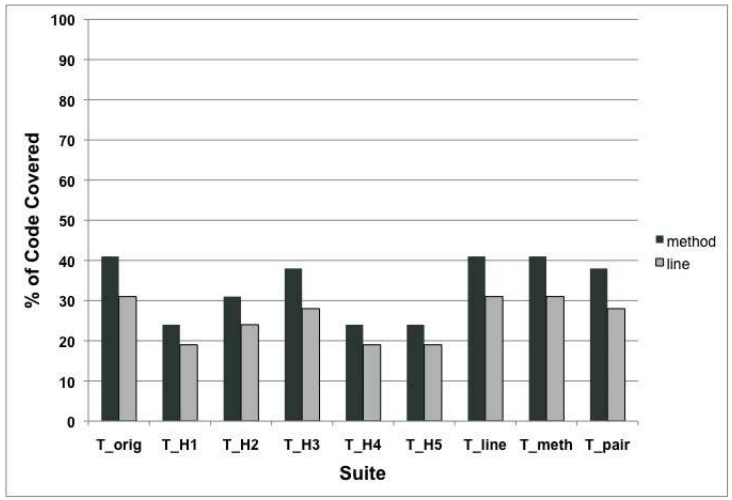


(d) jMSN

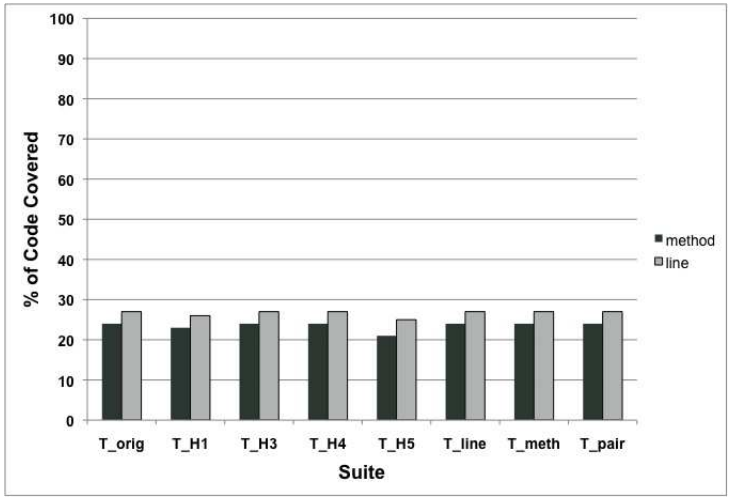
Figure 4.14: Cost and fault detection for each subject application's test suites



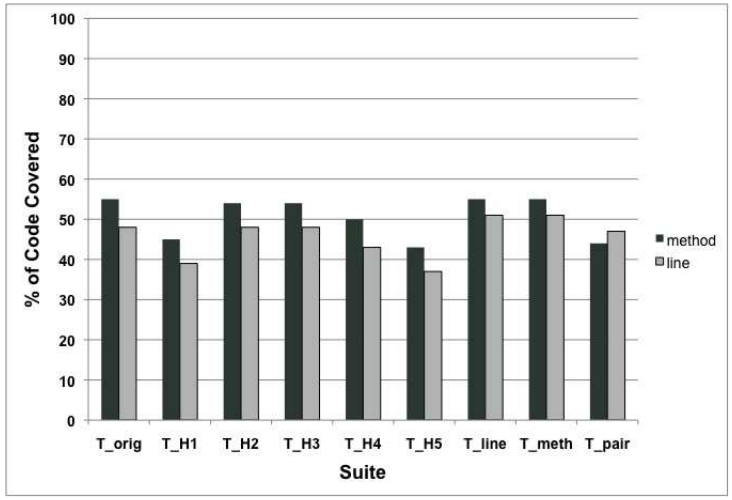
(a) CrosswordSage



(b) FreeMind



(c) jMSN



(d) GanttProject

Figure 4.15: Code coverage for all test suites

4.5 Discussion

This section presents further discussion on the results shown in the previous section.

4.5.1 Fault Detection

The fault detection effectiveness of CONTEST-generated test suites shows the validity of the model-based test case generation technique presented here. In fact, the number of faults detected by the generated test cases is 2.25 to 9.7 times the number of faults detected by the original suites. Combined with the cost savings of running and generating fewer test cases, the method presented here shows promise.

Additionally, for every application in this study, the CONTEST-generated test suites found faults that were not detected by the existing T_{orig} test suites. In CrosswordSage, T_{H3} found 4 faults that were not found by T_{orig}. In FreeMind, T_{H5} found 4 faults that T_{orig} did not find. In GanttProject, the T_{H5} test suite found 13 faults not discovered by T_{orig}. In jMSN, the T_{H2} test suite found 11 faults not found in the T_{orig} test suite. Conversely, FreeMind's T_{orig} suite found 5 faults missed by the consolidated suites (T_{Hn}) and jMSN's T_{orig} suite found 3 faults missed by the consolidated suites.

One of the generated test cases ($< e_{26}, e_{54}, e_{75}, e_{26}, e_{54} >$) detected Fault #6 in GanttProject. Figure 4.16 shows the relevant nodes in the graph and their conditional probabilities. This test case was generated using a history of 2. Specifically, the algorithm uses several highly likely pairs of events to generate the test case.

Interestingly enough, this sequence of events exists in T_{orig}, always followed by one more event, but did not detect Fault #6. It seems this is due to the GUI being placed in an unstable state by executing these five events without a sixth.

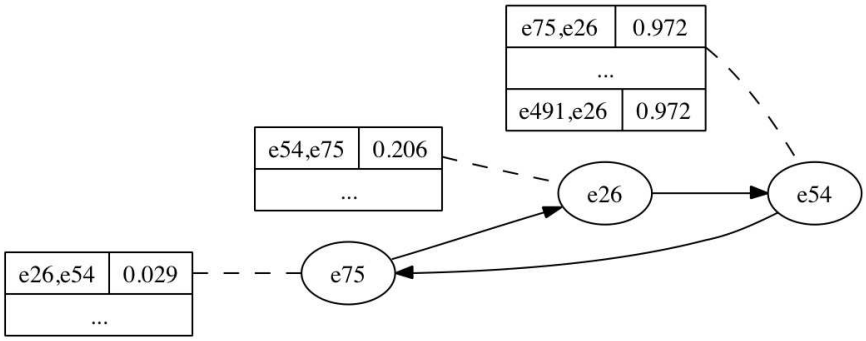


Figure 4.16: Tailored EFG Showing Generation of New Fault-Detecting Testcase

Comparatively, faults were missed by the generated test cases. An example of one is Fault #15 in jMSN, detected by a very short test case. The EFG for this test case is shown in Figure 4.17. The sequence of events $< e_5, e_6 >$ is less likely to occur in the input set of testcases ($P(e_6|e_5) = 0.008$), leading the algorithm to choose higher probability transitions when generating test cases.

4.5.2 Cost

Figure 4.14 shows a surprising result in relating cost of the test suite to fault detection. CrosswordSage's generated suites have similar fault detection regardless of suite size. In FreeMind, the smaller suites (T_{H2} and T_{H4}) detect the same number of faults and the larger suites (T_{H3} and T_{H5}) detect approximately the same number of faults. GanttProject's fault detection increases (and decreases) with suite size. jMSN's fault detection also correlates to the size of the test suite. These results

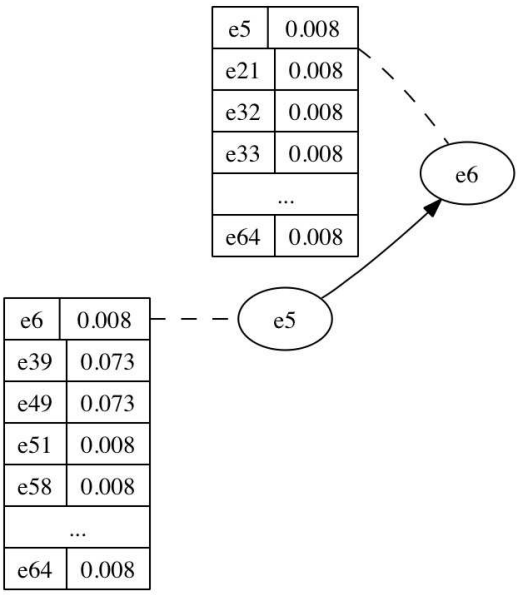


Figure 4.17: Tailored EFG Showing Existing Fault-Detecting Testcase

indicate that the size of the test suite does have an impact on fault detection.

4.5.3 Code Coverage

Differences in line coverage were also observed between the original suite and the generated suites. Specifically looking at the FreeMind application, T_{orig} covered lines not covered by T_{H2} and vice versa. Figures 4.18 and 4.19 show the tailored EFGs for both suites. Figure 4.18 shows one test case in T_{orig}, and Figure 4.19 shows a generated test case in T_{H2}. While the suites detected many of the same faults and had very similar line coverage, this shows an example of a generated test case covering different lines than the original.

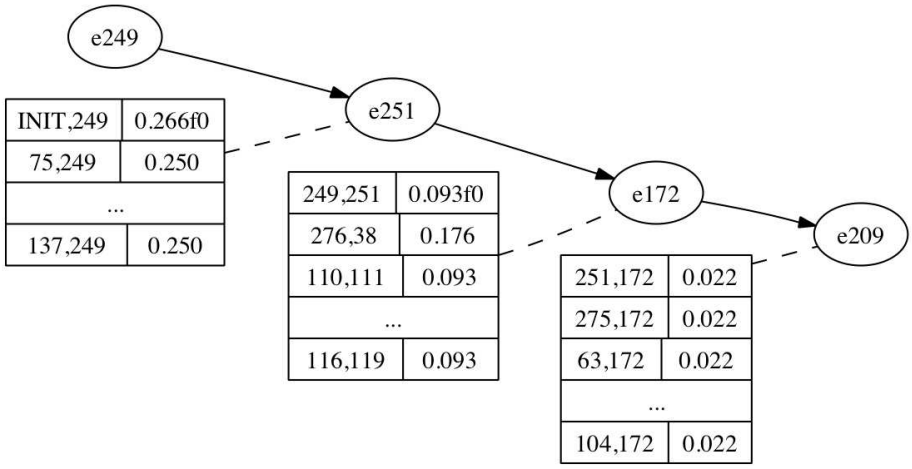


Figure 4.18: Tailored EFG for FreeMind test case in T_{orig}

4.6 Conclusions

This chapter introduced a probabilistic model and test case generation algorithm, CONTEST, based on the n-gram model. Using existing suites for four subject applications, generated suites (T_{H1}, T_{H2}, ... T_{H5}) were generally smaller, had better code coverage, and better fault detection than the original suite T_{orig}. The `history` parameter, used by the model to calculate probabilities of sequences of events, was tuned for each of the four subject applications used in the studies. In comparison to test suites reduced by method, line, and event pair coverage, the generated suites also performed better in terms of fault detection effectiveness and cost.

Comparing any one generated test suite to the input test suites, or defining a *similarity metric* for the event sequences, will provide further insight into why the generated test suites produce better results. Examining *event coverage*, whether all events in the input are exercised in the output, as well as examining *event sequence coverage* of event sequences of length `history`, are both of interest. Chapter 5 will

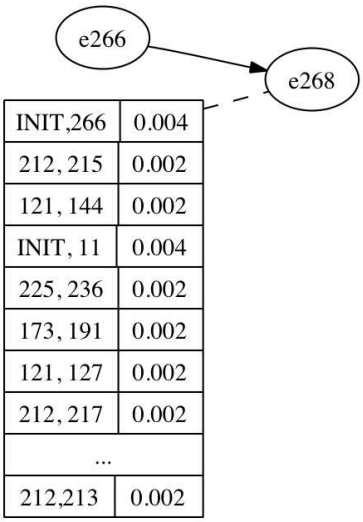


Figure 4.19: Tailored EFG for FreeMind test case in T_{H2}

further discuss these concepts.

Chapter 5

Introducing a Test Suite Similarity Metric

The previous chapter presented the results of a study which consolidated several test suites into one, based on a probabilistic model of the GUI and existing test suites. Previous research in software testing has yielded a large number of automated model-based test case generation techniques [6, 60, 39, 66, 25]. Each of these techniques has the ability to generate test suites containing hundreds of thousands of test cases, which require significant resources to run, and for regression testing, rerun [34]. For this reason, research in test case selection and reduction has been growing in an attempt to shrink these test suites to a manageable size, while maintaining the "goodness" of the original suite.

Reduction techniques attempt to yield a test suite that is "similar" to the original suite in some ways, where similarity is usually determined by using metrics based on code (*e.g.*, obtained from branch, line, and method coverage reports) executed by the original suite [46, 16, 30, 21, 48] or the set of faults detected. However, due to the nature of EDS systems and the influence of executing events in the *context* of previous events, another method of determining similarity is needed.

In this chapter, a new parameterized similarity metric, $CONTeSSi(n)$ (CONtext Test Suite Similarity) is defined. This metric explicitly considers the context of n preceding events in test cases to develop a new "context-aware" notion of test suite

similarity. This metric is an extension of the cosine similarity metric used in Natural Language Processing and Information Retrieval for comparing an item to a body of knowledge, *e.g.*, finding a query string in a collection of web pages or determining the likelihood of finding a sentence in a text corpus (collection of documents) [63, 1, 64]. $CONTeSSi(n)$ is evaluated by comparing the test suites used in Chapter 4. The results show that $CONTeSSi(n)$ is a better indicator of the similarity of test suites than existing metrics.

5.1 Computing Test Suite Similarity

The $CONTeSSi$ metric allows a tester to compare test suites while considering the context in which events are executed. This section presents the $CONTeSSi$ metric, describes the computation of the metric and provides an example of the application of the metric. The test suites used to test the Radio Button Demo GUI, shown in Figure 2.3 will be used again as a running example. Table 5.1, shown first in Table 1.1, is shown again here with an additional suite in the final column.

Original Tests	Event Pair Coverage	Event Coverage	Line Coverage	Method Coverage	Branch Coverage	Illustrative Tests
e_1, e_6	e_1, e_4	e_1, e_4	e_7, e_1, e_6	e_1, e_3, e_5, e_6	e_1, e_4	e_1
e_2, e_6	e_1, e_3, e_5, e_6	e_7, e_2, e_3, e_5, e_6	e_7, e_2, e_6	e_7, e_2, e_3	e_7, e_1, e_6	e_2
e_7, e_1	e_7, e_2, e_3, e_5, e_6		e_2, e_3, e_5, e_6		e_7, e_2, e_6	e_3
e_1, e_4	e_3, e_5, e_4, e_3, e_6		e_3, e_5, e_4, e_3, e_6		e_2, e_3, e_5, e_6	e_4
e_2, e_3, e_5	e_2, e_4, e_7, e_2, e_6		e_7, e_2, e_3, e_5, e_6			e_5
e_7, e_1, e_6	e_7, e_1, e_6					e_6
e_7, e_2, e_6						e_7
e_2, e_4 e_7, e_2, e_6						
e_7, e_2, e_3						
e_1, e_3, e_5, e_6						
e_2, e_3, e_5, e_6						
e_3, e_5, e_4, e_3, e_6						
e_7, e_2, e_3, e_5, e_6						

Table 5.1: Example test suites yielded from several reduction techniques

Suite	e_1	e_2	e_3	e_4	e_5	e_6	e_7
Original	5	8	7	3	5	9	6
Evnt Pair	3	3	4	3	3	5	3
Evnt Cov	1	1	1	1	1	1	1
Line Cov	1	3	4	1	3	5	3
Meth Cov	1	1	2	0	1	1	1
Brnch Cov	2	2	1	1	1	3	2
Illus. Suite	1	1	1	1	1	1	1

(a) Frequency of individual events

Suite	e_1, e_3	e_1, e_4	e_1, e_6	e_2, e_3	e_2, e_4	e_2, e_6	e_3, e_5	e_3, e_6	e_4, e_3	e_4, e_7	e_5, e_4	e_5, e_6	e_7, e_1	e_7, e_2
Original	1	1	2	4	1	3	5	1	1	1	1	3	2	4
Evnt Pair	1	1	1	1	1	1	3	1	1	1	1	2	1	2
Evnt Cov	0	1	0	1	0	0	1	0	0	0	0	1	0	1
Line Cov	0	0	1	2	0	1	3	1	1	0	1	2	1	2
Meth Cov	1	0	0	1	0	0	1	0	0	0	0	1	0	1
Brnch Cov	0	1	1	1	0	1	1	0	0	0	0	1	1	1
Illus. Suite	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b) Frequency of all event pairs

Suite	e_1, e_3, e_5	e_2, e_3, e_5	e_2, e_4, e_7	e_3, e_5, e_6	e_3, e_5, e_4	e_4, e_3, e_6	e_4, e_7, e_2	e_5, e_4, e_3	e_7, e_1, e_6	e_7, e_2, e_3	e_7, e_2, e_6
Original	1	3	1	3	1	1	1	1	1	2	2
Evnt Pair	1	1	1	2	1	1	1	1	1	1	1
Evnt Cov	0	1	0	1	0	0	0	0	0	1	0
Line Cov	0	2	0	2	1	1	0	1	1	1	1
Meth Cov	1	0	0	1	0	0	0	0	0	1	0
Brnch Cov	0	1	0	1	0	0	0	0	1	0	1
Illus. Suite	0	0	0	0	0	0	0	0	0	0	0

(c) Frequency of all event triples

Table 5.2: Frequency of n events in original and reduced test suites for Radio Button GUI example

By way of reminder, the suites in Table 5.1 are explained. Column 1 of Table 5.1 shows a test suite generated from the EFG model of the GUI. The test suites in the remaining columns were obtained using reduction techniques, and hence, are "similar" to the original suite. The popular HGS algorithm [16] was used to reduce the suite. Column 2 shows a suite reduced based on event pair coverage, which retains test cases that cover all unique pairs of events. The suite in Column 3 is reduced based on event coverage, which retains test cases that cover all unique events. The suites in columns 4-6 were reduced based on line, method, and branch coverage, respectively. Note that event-pair coverage is the only reduction method that considers the context of a preceding event; however, it considers only a single contextual event. Column 7 shows a suite consisting of test cases in which each test case executes one unique event. This suite will be used to illustrate the metric.

Although these suites are similar to the original suite in terms of their respective reduction/similarity criteria, they are quite different when considering context. For example, the subsequence $< e_7, e_2 >$ appears four times in the original suite; $< e_2, e_3 >$ appears four times; $< e_1, e_6 >$ appears twice; each of these event subsequences appear in multiple contexts. Reduction does not consider preserving the importance of these frequencies and/or contexts.

Next, consider some notions of the similarity of two given suites. Similarity can be measured based on the occurrence of events in both suites. If both suites contain exactly the same events, they could be considered to be very similar. However, that would also imply that a suite with 10 test cases of 5 events each, for a total of 50 distinct events, would be the same as a test suite with 1 test case with 50 of the same

events. A better method of measuring similarity, however, would be to consider the frequency of events in the test suite. For example, counting the occurrence of e_3 in each test suite and using this count to compare the suites will apply a weight to the event and provides more information on the suite. A vector is used to represent the count of each event in the suite, with each position in the vector representing the count of a single event. For the seven events e_1 to e_7 in the running example, this vector is produced: $< 5, 8, 7, 3, 5, 9, 6 >$, also shown in tabular form for all suites in Table 5.2(a). This is the basis of $CONTeSSi$.

Because EDS systems are highly reliant on the context in which events are executed, $CONTeSSi$ should return a value representing higher similarity when the same events occur in the same frequency and the same context between two test suites. As a starting point, consider the context for a single preceding event; a vector can be created based on the frequencies of event pairs observed in the test suite, rather than on a single event. In considering this context, the event pair coverage suite in Table 5.1 is expected to be more similar to the original suite than the event coverage suite, since the event pair coverage suite is created based on the existence of event pairs. Table 5.2(b) shows the count of each event pair for each suite. This is the basis of $CONTeSSi(n)$, for $n = 1$, since we are looking at events in the context of one other (previous) event.

Now, extending this example to compute $CONTeSSi(2)$, the frequencies shown in Table 5.2(c) are obtained. In general, as n increases, the frequencies for the event sequences decrease, as they appear less frequently in the test suites. Intuitively, comparing test suites on longer sequences will make it harder for the

test suites to be similar. Therefore, if two test suites have a high similarity score with a larger n, they are even more similar than two suites being compared with a small n. By treating each row in Table 5.2(a), (b), or (c) as a vector, $CONTeSSi$ is computed as follows:

$$CONTeSSi(A, B) = \frac{(A \cdot B)}{(|A| \times |B|)} \tag{5.1}$$

where A and B are the vectors corresponding to the two test suites, $A \cdot B$ is the dot product of the two vectors, *i.e.*, $\sum_{i=1}^{j}(A_i \times B_i)$ where j is the number of terms in the vector; and $|A| = \sqrt{\sum_{i=1}^{j}(A_i)^2}$. The value of $CONTeSSi$ lies between 0 and 1, where a value closer to 1 indicates more similarity. Hence, $CONTeSSi(n)$ is computed as shown in Equation 5.1, creating a vector for each suite, representing the frequencies of all possible groups of $n+1$ events. The inclusion of n previous events will increase the number of terms in the vector, thereby increasing j. The values in Table 5.3 show the values of $CONTeSSi(n)$ for all the test suites, for $n = 0, 1, 2, 3$. From these values, observe that if context is ignored, *i.e.*, use $n = 0$, most of the reduced suites are quite similar to the original, as indicated by the high (> 0.9) value of $CONTeSSi(0)$. However, the similarity between the test suites decreases as more context (larger values of n) is considered for the events. The event- and method-coverage suites show relatively lower values of $CONTeSSi(3)$ because they retain very little context with only two test cases. The event-pair reduced suites have the highest value of $CONTeSSi(3)$, followed by line and branch coverage reduced suites. Finally, the illustrative suite is very similar to the original when context is

not considered ($CONTeSSi(0) = 0.956$). With the addition of context, however, the $CONTeSSi$ value is 0 due to the single event test cases. Even in this simple example, the value of the similarity metric can be seen.

	Suite					
n	Evnt Pair	Evnt Cov	Line Cov	Meth Cov	Brnch Cov	Illus. Suite
0	0.977	0.956	0.970	0.921	0.960	0.956
1	0.969	0.869	0.967	0.859	0.946	0
2	0.963	0.813	0.960	0.794	0.931	0
3	0.959	0.774	0.952	0.754	0.923	0

Table 5.3: CONTeSSi(n) value for *Suite* compared to *Original* for all Radio Button GUI example suites

In order to improve the context information of events that appear at the beginning and end of test cases, two special sets of "events" called $INIT_n$ and $FINAL_n$ are included. Without loss of generality, these events are added to all test cases. When computing $CONTeSSi(n)$, n $INIT$ events are prepended and n $FINAL$ events are appended to each test case[1]. For example, in looking at event triples ($n = 2$), two $INIT$ and two $FINAL$ events are added to the sequences. In computing $CONTeSSi(2)$, a vector of the following format is used:

$$< INIT_0, INIT_1, e_2, e_3, e_5, FINAL_0, FINAL_1 >$$

to glean that e_2 is at the start of $< e_2, e_3, e_5 >$, by obtaining the triple $< INIT_0, INIT_1, e_2 >$ and that e_5 is at the end of the sequence by obtaining $< e_5, FINAL_0, FINAL_1 >$. These events are not shown in Table 5.2.

[1]Notice the sets of $INIT$ and $FINAL$ events are added to ensure CONTeSSi can look back n events. The $INIT$ and $FINAL$ nodes added to the EFG, as described in Section 4.1, are necessary to allow Markov reasoning on the EFG.

5.2 Empirical Study

To evaluate the quality of $CONTeSSi(n)$, an empirical study was conducted comparing nine test suites on the basis of exisiting similarity metrics, such as line coverage, method coverage, and event pair coverage. $CONTeSSi(n)$ was also used to compare the same suites.

The goal of this study is to *evaluate a metric that measures the similarity of two test suites and to determine the quality of this metric.* Restating this goal using the Goal Question Metric (GQM) Paradigm [2], the goal for this research is as follows:

> Analyze the **test suites**
> for the purpose of **comparison**
> with respect to **other test suites**
> from the point of view of the **tester/researcher**
> in the context of **event driven systems**.

From this goal, the following research questions are addressed:

RQ5.1 Is $CONTeSSi(n)$ a better indicator of similarity for larger values of n?

RQ5.2 Does $CONTeSSi(n)$ agree with existing metrics in determining the similarity between suites, specifically relating to fault detection effectiveness?

Each of these research questions will evaluate the similarity metric by comparing existing test suites on coverage criteria and the $CONTeSSi(n)$ metric. In most research and in practice, test suites are evaluated based on code coverage, fault detection, or both; the results of this study provide an objective method of comparing test suites without the need to run them.

The first question is focused on comparing the results of $CONTeSSi(n)$ to the coverage of the suite, and further examining the relationship between the metric and fault detection. The second question recognizes the importance of event context in EDS test cases. By varying the amount of event context used in computing the metric, a finer grained measure of the similarity between test suites is garnered.

In setting up this study, several subject applications were chosen, test suites were developed and run, and the suites were compared based on several metrics. Each of these actions are described in the following sections.

5.2.1 Subject Applications, Tools, and Test Suites

The subject applications and tools for this study are the same as those described in the previous chapter (Sections 4.3.1 and 4.3.2). This empirical study used the same test suites as the CONTEST study (Section 4.3) as a basis for determining the usefulness of the test suite similarity metric. As a reminder, the suites consisted of: one model-generated suite (T_{orig}), five CONTEST-generated suites ($T_{H1}, T_{H2}, ..T_{H5}$), and three reduced suites (T_{line}, T_{method}, T_{pair}).

5.2.2 Procedure

Figure 5.1 gives a graphical representation of the steps described here. First, test suites based on the EFG were created using the parameterized test case generator. Next, the test suites were executed using GUITAR's JavaGUIReplayer. After running the test suites, fault detection and code coverage was collected for each test

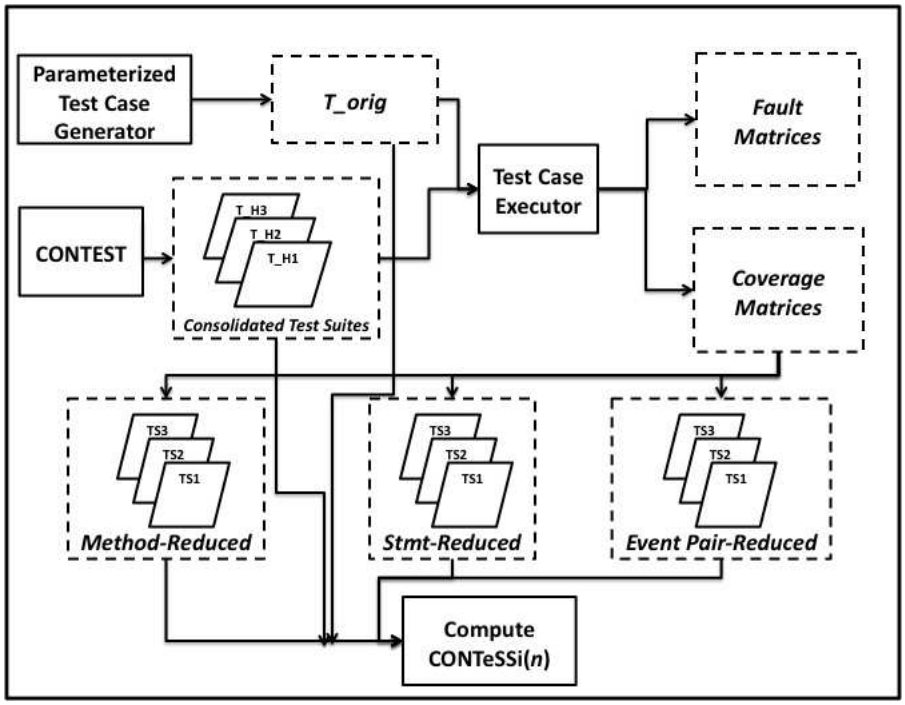


Figure 5.1: Comparing test suites using the CONTeSSi metric

case, matrices were built, and this information was used to compare the suites. The test oracle used for this work detects crashes in these applications, where a crash is defined as an uncaught exception thrown during test case execution.

The generated test cases were also used as input to CONTEST, as in Chapter 4. Five test suites were obtained for each application, with `history` values of 1 through 5. These suites are shown in Figure 5.1 as T_Hn where n is the value of `history` used.

From the coverage matrices, reduced suites were obtained based on event pair, method and line coverage. The code coverage and fault detection of these reduced suites was computed from the per-test-case coverage files generated during the execution of the original suite.

Finally, a file was created for each test suite where each line of the file rep-

resented one test case and contained the sequence of events in a test case. Using these files as input, $CONTeSSi(n)$ was computed, using values for n from 0 to 5, to compare the original suite to each reduced suite.

5.2.3 Comparing Test Suites

Using Equation 5.1, the $CONTeSSi$ metric is computed. To use this metric, it is also necessary to have a method of comparing suites with other metrics. The following function can be used to compare two suites given any of the metrics discussed here.

$$f(T_{orig}, T, m) = \frac{N(e_m(T_{orig}) \bigcap e_m(T))}{N(e_m(T_{orig}))} \tag{5.2}$$

where T is the suite being compared to the original suite, m is one of the metrics on which suites are compared, such as line coverage, branch coverage or event pair coverage, $e_m(suite)$ is a function returning the set of elements for metric m covered by *suite*, and N is a function returning the number in the set given. The result of this function is a number between 0 and 1 which represents the ratio of the number of *metric* elements covered by both T and T_{orig} to the total number of *metric* elements covered by T_{orig}. Coverage and fault detection numbers are shown for each suite; the suites are then compared based on these metrics.

5.2.4 Threats to Validity

There are a few threats to validity which should be considered when interpreting the results of this study. First, due to a desire to use the existing GUITAR infrastructure, and to compare the results to those posted by previous graduate student researchers, subject applications developed in Java were used. Therefore, this study presents no information on how the results would translate to other development languages. Second, this research is concerned only with EDS systems; this method may not be appropriate for test suites in other domains.

Third, although each application is different, they do not reflect all possible classes of EDS. Fourth, the majority of the application code is written for the GUI, meaning the results may not be consistent for applications with a simple GUI and complex underlying business logic.

Fifth, a potential problem for this study is that it may not produce conclusive results on which value of n used for the context of $CONTeSSi(n)$ is most effective; however, it does give an indication of the impact of context and a trend of the results as the value of n is varied.

5.3 Results

This section presents the value returned by $CONTeSSi(n)$ for the test suites used in this study. The traditional metrics are shown for each suite as a benchmark for comparison. Ultimately, the value of $CONTeSSi(n)$ is compared to the traditional metrics using the function described in Section 5.2.3.

5.3.1 CONTeSSi with Increasing Values of n

RQ5.1: Is $CONTeSSi(n)$ a better indicator of similarity for larger values of n?

Metrics: $CONTeSSi(n)$ *value for* $n = 0..5$

Computing $CONTeSSi(n)$ without context (for $n = 0$), the data in Table 5.4 shows that at least one of the CONTEST suites is the most similar to the original suite, T_{orig}. For CrosswordSage, CONTEST's T_{H3}, T_{H4} and T_{H5} are almost identical to each other and the most similar to T_{orig}; T_{H1} and T_{H2} are less similar to T_{orig}. For FreeMind, T_{H3} and T_{H4} are the most similar to T_{orig}, closely followed by T_{H5}. For GanttProject, T_{H3} and T_{H4} are the most similar to T_{orig}, followed by T_{H2}. T_{H1} is less similar and T_{H5} is very dissimilar. For jMSN, T_{H2} and T_{H3} are the same and very similar to T_{orig}, closely followed by T_{H4}. T_{H5} is less similar and T_{H1} is the most dissimilar.

The reduced suites also have interesting results. In three of the four applications, the T_{pair} suites are more similar than the other reduced suites. For the fourth application, FreeMind, the T_{method} suite is the most similar reduced suite, followed by T_{pair} and T_{line}.

As the value of n increases, $CONTeSSi(n)$ decreases in *most* cases, indicating a decrease in the similarity between the two suites as more context is considered, as expected from the example in Table 5.3. While there are some values of n, that cause the $CONTeSSi(n)$ value to increase, the difference is so slight that it is unclear whether or not this result is significant.

		Suite							
Application	n	T_{H1}	T_{H2}	T_{H3}	T_{H4}	T_{H5}	T_{method}	T_{pair}	T_{line}
CrosswordSage	0	0.979	0.976	1.000	1.000	0.999	0.788	0.974	0.760
	1	0.965	0.960	0.996	0.996	0.996	0.682	0.958	0.676
	2	0.958	0.951	0.991	0.990	0.990	0.597	0.952	0.614
	3	0.951	0.944	0.984	0.984	0.983	0.539	0.946	0.579
	4	0.944	0.937	0.977	0.977	0.977	0.496	0.942	0.552
	5	0.937	0.931	0.970	0.970	0.970	0.464	0.938	0.528
FreeMind	0	0.991	0.904	1.000	0.999	0.996	0.105	0.087	0.064
	1	0.986	0.911	0.997	0.997	0.994	0.088	0.079	0.051
	2	0.981	0.906	0.992	0.992	0.989	0.089	0.084	0.051
	3	0.975	0.901	0.986	0.986	0.983	0.091	0.089	0.162
	4	0.969	0.895	0.980	0.980	0.977	0.093	0.093	0.055
	5	0.963	0.890	0.975	0.974	0.971	0.095	0.098	0.057
GanttProject	0	0.888	0.923	1.000	0.999	0.539	0.200	0.947	0.358
	1	0.875	0.897	0.996	0.995	0.611	0.237	0.934	0.423
	2	0.871	0.892	0.991	0.991	0.610	0.229	0.933	0.414
	3	0.866	0.888	0.987	0.986	0.607	0.224	0.931	0.410
	4	0.862	0.884	0.982	0.981	0.604	0.220	0.930	0.408
	5	0.858	0.879	0.977	0.977	0.601	0.216	0.929	0.406
jMSN	0	0.753	0.997	0.997	0.956	0.888	0.233	0.612	0.491
	1	0.730	0.968	0.968	0.931	0.864	0.172	0.603	0.389
	2	0.704	0.935	0.935	0.899	0.830	0.146	0.615	0.355
	3	0.681	0.904	0.904	0.870	0.803	0.131	0.626	0.331
	4	0.661	0.877	0.877	0.843	0.778	0.121	0.635	0.312
	5	0.642	0.852	0.852	0.819	0.756	0.114	0.643	0.298

Table 5.4: CONTeSSi(n) for $T_{orig}, Suite$

5.3.2 CONTeSSi vs. Traditional Metrics

RQ5.2: Does $CONTeSSi(n)$ agree with existing metrics in determining the similarity between suites, specifically relating to fault detection effectiveness?

Metrics: *Line coverage, Method coverage, Pair coverage*

To determine if the $CONTeSSi$ metric returns a value consistent with the "goodness" of a suite, the computation of $CONTeSSi(n)$ shown in Table 5.4 and the traditional metrics shown in Table 5.5, combined with the fault detection reported in Figure 4.12 are used to determine the most effective method of detecting test

suite similarity.

Application	Suite	Coverage Value		
		Class	Method	Block
CrosswordSage	T_{orig}	41	20	25
	T_{H1}	47	20	20
	T_{H2}	41	20	20
	T_{H3}	41	20	20
	T_{H4}	41	20	20
	T_{H5}	82	47	55
	T_{method}	35	15	23
	T_{pair}	41	20	25
	T_{line}	35	20	25
FreeMind	T_{orig}	55	32	26
	T_{H1}	49	26	21
	T_{H2}	51	28	23
	T_{H3}	55	33	26
	T_{H4}	53	30	24
	T_{H5}	55	32	26
	T_{method}	51	29	24
	T_{pair}	55	32	26
	T_{line}	49	26	23
GanttProject	T_{orig}	66	51	46
	T_{H1}	58	44	46
	T_{H2}	58	43	45
	T_{H3}	58	44	46
	T_{H4}	58	44	46
	T_{H5}	58	43	45
	T_{method}	58	44	45
	T_{pair}	58	44	46
	T_{line}	66	50	45
jMSN	T_{orig}	35	24	27
	T_{H1}	35	23	27
	T_{H2}	35	24	27
	T_{H3}	35	24	27
	T_{H4}	35	24	27
	T_{H5}	34	21	26
	T_{method}	28	16	20
	T_{pair}	35	24	27
	T_{line}	35	24	27

Table 5.5: Code Coverage Information

Extending the trends and relationships of Table 5.4 to the faults detected in each suite (Figure 4.12), it can be seen that the values returned by $CONTeSSi(n)$ are consistent with the faults detected by the suites. That is, for every application, the CONTEST suites which returned faults most similar to the original suite also received the highest $CONTeSSi$ scores. A similar result was found for the reduced suites. That is, for every application, T_{pair} detected almost the same faults as the original suite, while T_{method} and T_{line} detected fewer.

Traditional code coverage metrics are shown in Table 5.5. For all four appli-

cations, the class, method, and block coverage of the reduced suites are very similar to the original suite. Using this metric as a gauge for test suite similarity would lead a tester to believe the suites are very similar; however, the fault detection of each suite indicates otherwise. This finding supports the intuition described earlier that traditional metrics are not a good measure of similarity between test suites.

		Suite							
Application	Metric	T_{H1}	T_{H2}	T_{H3}	T_{H4}	T_{H5}	T_{method}	T_{pair}	T_{line}
CrosswordSage	`method`	1	1	1	1	1	1	1	1
	`pair`	0.088	0.967	0.967	0.921	0.921	0.016	1	0.022
	`line`	0.969	1	1	0.992	0.992	0.477	0.496	1
FreeMind	`method`	0.169	0.714	0.810	0.762	0.857	1	0.749	0.479
	`pair`	0.010	0.784	0.784	0.582	0.357	0.006	1	0.006
	`line`	0.494	0.850	0.939	0.890	1	0.974	0.759	1
GanttProject	`method`	0.861	0.997	0.997	0.884	0.821	1	0.999	0.968
	`pair`	0.022	1	1	1	0.110	0.003	1	0.003
	`line`	0.836	0.980	0.980	0.902	0.809	0.970	0.978	1
jMSN	`method`	0.971	0.993	0.993	0.993	0.893	1	1	0.200
	`pair`	0.011	0.803	0.711	0.711	0.016	0.347	1	0.008
	`line`	0.977	0.999	0.998	0.998	0.969	0.989	1	1

Table 5.6: Computing $f(T_{orig}, T, metric)$

Table 5.6 shows the computation of Equation 5.2 for each metric, for each application. Each combination of metric and test suite are considered *i.e.*, the number of methods covered by each of the test suites are counted for each application. For almost every metric, the CONTEST suite covers the most elements of the metric. The best CONTEST suite differs with each application; the results are consistent with the suite with the best fault detection effectiveness. For CrosswordSage, all of the suites yield a value of 1 for `method` coverage. T_{H2} and T_{H3} have the best `pair` and `line` coverage. For FreeMind, T_{H5} yields the best `method` coverage. T_{H2} and T_{H3} have the best `pair` coverage. T_{H5} yields a value of 1 for `line` coverage, followed by T_{H3}. For GanttProject, T_{H2} and T_{H3} are tied for the best `method` coverage;

closely behind T_{pair}. T_{H2}, T_{H3}, and T_{H4} are tied for the best `pair` coverage. T_{H2} and T_{H3} also have the best values for `line` coverage. For jMSN, T_{H2}, T_{H3}, and T_{H4} show the best `method` coverage, slightly behind T_{pair}. T_{H2} has the best values for `pair` and `line` coverage. Fault detection effectiveness of each suite further confirms this ranking of the suites.

5.4 Discussion

The similarity (or rather dissimilarity) between FreeMind's original suite and reduced suites does not follow the pattern of the other applications. This can be partially explained by the redundancy within test cases in the original suite, combined with the fact that the computation of $CONTeSSi(n)$ counts events (or event sequences). Because much of the redundancy is removed when the suites are reduced, the number of test cases as well as the counts of events (or event sequences) used in computing $CONTeSSi$ are much smaller. Additionally, these reduced suites did not find many faults; T_{pair}, however, had better fault detection than the others.

By comparing the test suites on existing metrics, which were also used to create the suites (Table 5.6), some insight into the value of these reduction techniques is gained. For all four applications, the similarity of the CONTEST suites to T_{orig}, measured in the ratio of elements covered, code coverage, and fault detection, also strengthens the claim that context is valuable in EDS test cases. This comparison also serves to reinforce the results provided by $CONTeSSi$ on the similarity and "goodness" of each suite.

$CONTeSSi(n)$ was designed with context in mind due to the importance of context in test cases for EDS. It is interesting to note the trend of the $CONTeSSi(n)$ value. In some suites for some of the applications, the value of $CONTeSSi(n+1)$ increases over $CONTeSSi(n)$ rather than decreasing as is the general overall trend. For example, the $CONTeSSi(1)$ value for GanttProject's T_{line} suite is larger than that of $CONTeSSi(0)$. The remaining $CONTeSSi$ values decrease, however, as n increases. Conversely, FreeMind and jMSN's T_{pair} values of $CONTeSSi$ for $n > 1$ increase as context is increased. It is possible this is due to the length of the test cases; as n gets closer to the length of the test case, the similarity between the suites increases.

5.5 Conclusions

Although there are several existing techniques (*e.g.*, reduction and minimization) used to obtain test suites that are "similar" to an original suite, existing techniques are not well suited to EDS. This chapter presented a new parameterized metric called $CONTeSSi(n)$, which uses the context of n preceding events in test cases to quantify test suite similarity for EDS. $CONTeSSi(n)$ is appropriate for EDS because it considers the contextual relationships between events, proven to be important in testing EDS. This metric was defined for and evaluated on eight test suites for four open source applications. These results showed that $CONTeSSi(n)$ is a better indicator of the similarity of EDS test suites than existing metrics.

Chapter 6

Conclusions and Future Research Directions

This chapter will describe conclusions of the studies performed as part of this research, followed by some possible future research directions opened up by this work.

6.1 Conclusions

The research presented in this document focused on proving the following thesis statement: **A method of combining and consolidating sequence-based test suites preserves the context observed in the existing test suites and maintains their fault detection effectiveness.**

Chapter 3 presented a study of the effectiveness of crash testing for industrial systems with a GUI front-end. One of the most interesting findings in this study is the high percentage of defects found through the GUI that are actually defects in the underlying business logic of the system. While this has been shown in open source systems in past research [6, 39, 61, 66], it is interesting that this finding holds in testing industry systems as well. It is another indication that it is important to perform GUI testing, both to test the system through the GUI and to test the GUI itself.

The results of this study further show the correlation between the test suite

design and the defects detected by crashes. The study also reinforced the idea that the underlying code is often tested through the GUI due to the window it provides into the system behavior. However, the results of this study also show that more visibility is needed so that more of the defects currently detected only by user observation can be detected before the system is released to the field.

The CONTEST algorithm presented in Chapter 4, based on a probabilistic model representation of the GUI used to generate test cases, provides a method of consolidating existing test suites. Populating the model with existing sets of sequences, in the form of usage profiles and test cases generated with the parameterized test case generation algorithm allowed the demonstration of the model's usefulness in generating new test cases based on the probability of event sequences gleaned from the input set of event sequences.

The CONTEST study relied on crashes to find defects in the applications, encouraged by the results given in Chapter 3. Again, the results of the study show crash testing is useful in finding defects. Specifically, crashes for this study are in the form of *uncaught exceptions*. There are other kinds of crashes that can occur in a running application, and further study may be warranted.

$CONTeSSi(n)$, the test suite similarity metric presented in Chapter 5, provides a method of comparing existing test suites for EDS. This is the first metric which considers the context of event execution as part of its computation. Borrowing the cosine similarity metric from the IR and NLP fields, it was tailored to be appropriate for EDS test suites. This metric provides a better indication of similarity of these suites than existing, usually code-based, metrics.

Each of the studies presented in Chapters 4 and 5 can benefit from expanding to more subject applications, and specifically, to industrial subject applications. Further, expanding to other types of EDS systems, such as web applications, will also increase the confidence in the results presented here. Combining the model-based test case consolidation presented in Chapter 4 with the $CONTeSSi(n)$ metric, the CONTEST test case generation algorithm can be reworked to consider the output of $CONTeSSi$ as input to the stopping criteria.

6.2 Future Research

The study presented in Chapter 3 is an initial characterization of GUI systems, and several steps succeed it. This first study looked at three systems driven by system events. In future work, another class of systems could be studied, such as those driven by user interactions. After further characterizing GUI systems based on the same criteria presented here, it may then be possible to develop a methodology for generating test strategies based on the characterization. Additionally, the findings presented here can be applied to the software development process for other software development groups to determine the impact of GUI system characterization on the effectiveness of testing for future releases.

The CONTEST study presented in Chapter 4 also offers opportunities for future work. The model-based approach can be extended to perform simulated regression testing, using two or more fielded versions of an open source application. Using existing event sequences from one version and using them to populate a model

of a subsequent version will require some extensions to the model in order to map from one EFG to the other, and is a planned extension of this research. By using applications which are already fielded, it is possible to get immediate feedback, without waiting for an application to be developed.

Additionally, there are several possible modifications to the test case generation algorithm that will be explored. Currently, the algorithm generates test cases based on event sequences that contain at least one highly probable n-tuple of events, but the probability of the whole event sequence, and therefore the test case, may be very low. For example, if $P(e_2|e_1) = 0.999$, the algorithm will construct a test case that contains the sequence $< e_1, e_2 >$, even if e_1 is only exercised in 0.001% of the input sequences. In the future, techniques which consider the probability of a whole sequence of events, rather than just the n-gram, will be used in generating a test case. By allowing the user to specify a threshold probability, the case of sequences with a very low probability being chosen for test case generation will be avoided. Another variation of the algorithm is to traverse the least likely paths in the model to reveal rarely-encountered faults that may otherwise be difficult to detect. Adding the ability to provide constraints such as "*Execute an* Open *or* New *before a* Save" as input to the model will also be examined in future versions of the algorithm.

The results shown in Chapter 5 have also created several opportunities for future work. In the short term, the study can be extended to include additional subjects to reduce threats to external validity. In the medium term, $CONTeSSi(n)$ can be used to develop a new reduction technique for GUI test suites. The expectation is that the reduced suite will be better at retaining the fault detection

effectiveness of the original suite. The relationship between the test case length and the value of n used in $CONTeSSi(n)$ can be further investigated to draw conclusions regarding the behavior witnessed in two of the subject applications.

Bibliography

[1] Aslam, J. A., and Frost, M. An information-theoretic measure for document similarity. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval* (New York, NY, USA, 2003), ACM, pp. 449–450.

[2] Basili, V. R. Software modeling and measurement: the goal/question/metric paradigm. Tech. rep., University of Maryland at College Park, 1992.

[3] Beizer, B. *Software Testing Techniques*. Van Nostrand Reinhold Co., 1990.

[4] Binder, R. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[5] Brooks, P., Robinson, B., and Memon, A. M. An initial characterization of industrial graphical user interface systems. In *ICST '09: Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation* (2009).

[6] Brooks, P. A., and Memon, A. M. Automated GUI testing guided by usage profiles. In *ASE '07: Proc. of the 22nd IEEE/ACM Int'l Conf on Automated Software Engineering* (New York, NY, USA, 2007), ACM, pp. 333–342.

[7] Brooks, P. A., and Memon, A. M. Introducing a test suite similarity metric for event sequence-based test cases. In *ICSM '09: Proc. of the 25th IEEE Int'l Conf on Software Maintenance* (2009).

[8] Butcher, M. Munro, H., and Kratschmer, T. Improving software testing via odc: Three case studies. *IBM Systems Journal 41*, 1 (2002), 31–44.

[9] Chillarege, R. Orthogonal defect classification. In *Handbook of Software Reliability and System Reliability*, M. R. Lyu, Ed. McGraw-Hill, Hightstown, NJ, 1996, pp. 359–400.

[10] Clarke, J. M. Automated test generation from a behavioral model. In *Proc. of the Eleventh Int'l Software Quality Week* (May 1998).

[11] Cripps, A., and Nguyen, N. Fuzzy lattice neurocomputing using weighted cosine similarity measure. *Neural Networks, 2007. IJCNN 2007. International Joint Conference on* (Aug. 2007), 236–241.

[12] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. Model-based testing in practice. In *Proc. of the 21st Int'l Conf on Software engineering* (1999), IEEE Computer Society Press, pp. 285–294.

[13] Finsterwalder, M. Automating acceptance tests for GUI applications in an extreme programming environment. In *Proc. Second Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng.* (2001), pp. 114–117.

[14] Hammontree, M. L., Hendrickson, J. J., and Hensley, B. W. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In *Proc. of the SIGCHI Conf on Human factors in computing systems* (1992), ACM Press, pp. 431–432.

[15] Hao, D., Pan, Y., Zhang, L., Zhao, W., Mei, H., and Sun, J. A similarity-aware approach to testing based fault localization. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (New York, NY, USA, 2005), ACM, pp. 291–294.

[16] Harrold, M. J., Gupta, R., and Soffa, M. L. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol. 2*, 3 (1993), 270–285.

[17] Harrold, M. J., Gupta, R., and Soffa, M. L. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol. 2*, 3 (1993), 270–285.

[18] Heimdahl, M., and George, D. Test-suite reduction for model based tests: effects on test quality and implications for testing. *Automated Software Engineering, 2004. Proceedings. 19th International Conference on* (Sept. 2004), 176–185.

[19] Hicinbothom, J., and Zachary, W. A tool for automatically generating transcripts of human-computer interaction. In *Proc. Human Factors and Ergonomics Society 37th Ann. Meeting* (1993), p. 1042.

[20] IEEE. *IEEE standard classification for software anomalies. IEEE Standard 1044-1993*, Jun 1994.

[21] Jones, J., and Harrold, M. Test-suite reduction and prioritization for modified condition/decision coverage. *Software Engineering, IEEE Transactions on 29*, 3 (March 2003), 195–209.

[22] JUnit. Testing resources for extreme programming. http://junit.org/news/extension/gui/index.htm, 2004.

[23] Kaner, C., Falk, J., and Nguyen, H. *Testing Computer Software*, 2nd ed. Van Nostrand Reinhold, 1993.

[24] Karnik, A., Goswami, S., and Guha, R. Detecting obfuscated viruses using cosine similarity analysis. *Modelling & Simulation, 2007. AMS '07. First Asia International Conference on* (March 2007), 165–170.

[25] Kasik, D. J., and George, H. G. Toward automatic generation of novice user test scripts. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 1996), ACM, pp. 244–251.

[26] Kilgarriff, A. Comparing corpora. *International Journal of Corpus Linguistics 6*, 1 (2001), 1–37.

[27] Kilgarriff, A., and Grefenstette, G. Introduction to the special issue on the web as corpus. *Comput. Linguist. 29*, 3 (2003), 333–347.

[28] Knuth, D. E. The errors of tex. *Software-Practice and Experience 19*, 7 (1989), 607–685.

[29] Lau, M. F., and Yu, Y. T. An extended fault class hierarchy for specification-based testing. *ACM Trans. Software Engineering Methodology 14*, 3 (2005), 247–276.

[30] Leon, D., and Podgurski, A. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on* (Nov. 2003), 442–453.

[31] Li, N., Francis, P., and Robinson, B. Static detection of redundant test cases: An initial study. In *ISSRE* (2008), pp. 303–304.

[32] Marick, B. When should a test be automated? In *Proc. 11th Int'l Software/Internet Quality Week* (1998).

[33] Marick, B. Bypassing the GUI. *Software Testing and Quality Engineering Magazine* (2002), 41–47.

[34] McMaster, S., and Memon, A. Call-stack coverage for gui test suite reduction. *IEEE Transactions on Software Engineering 34*, 1 (2008), 99–115.

[35] Memon, A. *A Comprehensive Framework for Testing Graphical User Interfaces*. PhD thesis, Dept. of Computer Science,Univ. of Pittsburgh, 2001.

[36] Memon, A. M. Employing user profiles to test a new version of a GUI component in its context of use. *Software Quality Control 14*, 4 (2006), 359–377.

[37] Memon, A. M., Banerjee, I., and Nagarajan, A. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proc. of the 10th Working Conf on Reverse Engineering* (2003), pp. 260–269.

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

[39] Memon, A. M., and Xie, Q. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering 31*, 10 (2005), 884–896.

[40] Myers, B. A. User interface software tools. *ACM Trans. Comput.-Hum. Interact. 2*, 1 (1995), 64–103.

[41] Nagarajan, A., and Memon, A. M. Refactoring using event-based profiling, 2003.

[42] Ostrand, T., Anodide, A., Foster, H., and Goradia, T. A visual test development environment for GUI systems. In *Proc. of the 1998 ACM SIGSOFT Int'l symposium on Software testing and analysis* (1998), ACM Press, pp. 82–92.

[43] Ostrand, T. J., and Weyuker, E. J. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software 4*, 4 (1984), 289–300.

[44] Özekici, S., Altinel, I. K., and Angün, E. A general software testing model involving operational profiles. *Probab. Eng. Inf. Sci. 15*, 4 (2001), 519–533.

[45] Press, L. Personal computing: Windows, dos and the mac. *Commun. ACM 33*, 11 (1990), 19–26.

[46] Qu, X., Cohen, M. B., and Woolf, K. Combinatorial interaction regression testing: A study of test case generation and prioritization. *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on* (Oct. 2007), 255–264.

[47] Robinson, B., Francis, P., and Ekdahl, F. A defect-driven process for software quality improvement. In *2nd Intl Symposium on Empirical Software Eng and Measurement* (October 2008), pp. 333–335.

[48] Rothermel, G., Harrold, M., Ostrin, J., and Hong, C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. *Software Maintenance, 1998. Proceedings. International Conference on* (Nov 1998), 34–43.

[49] Schneider, G. M., Martin, J., and Tsai, W. T. An experimental study of fault detection in user requirements documents. *ACM Trans. on Software Engineering Methodology 1*, 2 (1992), 188–204.

[50] Shehady, R. K., and Siewiorek, D. P. A method to automate user interface testing using variable finite state machines. In *FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)* (Washington, DC, USA, 1997), IEEE Computer Society, p. 80.

[51] Sprenkle, S., Gibson, E., Sampath, S., and Pollock, L. Automated replay and failure detection for web applications. In *ASE '05: Proc. of the 20th IEEE/ACM Int'l Conf on Automated software engineering* (New York, NY, USA, 2005), ACM Press, pp. 253–262.

[52] Vijayaraghavan, G., and Kaner, C. Bugs in your shopping cart. http://www.testingeducation.org/a/bsct.pdf.

[53] Walton, G., Poore, J., and Trammell, C. Statistical testing of software based on a usage model. *Software: Practice and Experience 25*, 1 (Jan 1995), 97–108.

[54] White, L., AlMezen, H., and Alzeidi, N. User-based testing of GUI sequences and their interactions. In *Proc. 12th Int'l Symposium on Software Reliability Engineering* (2001), pp. 54–63.

[55] Whittaker, J., and Poore, J. Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology 2*, 1 (Jan 1993), 93–106.

[56] Whittaker, J. A., and Thomason, M. G. A Markov chain model for statistical software testing. *IEEE Trans. Softw. Eng. 20*, 10 (1994), 812–824.

[57] Woit, D. Conditional-event usage testing. In *CASCON '98: Proc. of the 1998 Conf of the Centre for Advanced Studies on Collaborative research* (1998), IBM Press, p. 23.

[58] Woit, D. M. Specifying operational profiles for modules. In *ISSTA '93: Proc. of the 1993 ACM SIGSOFT Int'l Symposium on Software Testing and Analysis* (1993), ACM Press, pp. 2–10.

[59] Wong, W. E., Horgan, J. R., London, S., and Mathur, A. P. Effect of test set minimization on fault detection effectiveness. In *ICSE '95: Proceedings of the 17th international conference on Software engineering* (New York, NY, USA, 1995), ACM, pp. 41–50.

[60] Xie, Q., and Memon, A. Automated model-based testing of community-driven open source GUI applications. In *Proc. 22nd Int'l Conf on Software Maintenance* (Sep 2006).

[61] Xie, Q., and Memon, A. M. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology 16*, 1 (2007), 4.

[62] Xie, Q., and Memon, A. M. Using a pilot study to derive a gui model for automated testing. *ACM Trans. Softw. Eng. Methodol. 18*, 2 (2008), 1–35.

[63] Y, T. R., and Kilgarriff, A. Measures for corpus similarity and homogeneity. In *Proceedings of the 3rd conference on Empirical Methods in Natural Language Processing* (1998), ACL-SIGDAT, pp. 46–52.

[64] Yilmaz, E., Aslam, J. A., and Robertson, S. A new rank correlation coefficient for information retrieval. In *SIGIR '08: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval* (New York, NY, USA, 2008), ACM, pp. 587–594.

[65] Yu, Y., Jones, J. A., and Harrold, M. J. An empirical study of the effects of test-suite reduction on fault localization. In *ICSE '08: Proceedings of the 30th international conference on Software engineering* (New York, NY, USA, 2008), ACM, pp. 201–210.

[66] Yuan, X., and Memon, A. M. Using GUI run-time state as feedback to generate test cases. In *ICSE'07, Proc. of the 29th Int'l Conf on Software Engineering* (May 23–25, 2007).