GUI Interaction Testing: Incorporating Event Context

Xun Yuan, *Member, IEEE,* Myra B. Cohen, *Member, IEEE,* and Atif M Memon, *Member, IEEE*

Abstract—Graphical user interfaces (GUIs), due to their event driven nature, present an enormous and potentially unbounded way for users to interact with software. During testing it is important to “adequately cover” this interaction space. In this paper, we develop a new family of coverage criteria for GUI testing grounded in combinatorial interaction testing. The key motivation of using combinatorial techniques is that they enable us to incorporate “context” into the criteria in terms of event combinations, sequence length, and by including all possible positions for each event. Our new criteria range in both efficiency (measured by the size of the test suite) and effectiveness (the ability of the test suites to detect faults). In a case study on eight applications, we automatically generate test cases and systematically explore the impact of context, as captured by our new criteria. Our study shows that by increasing the event combinations tested and by controlling the relative positions of events defined by the new criteria, we can detect a large number of faults that were undetectable by earlier techniques.

Index Terms—GUI testing, automated testing, model-based testing, combinatorial interaction testing, GUITAR testing system.

1 Introduction

An important characteristic of graphical user interfaces (GUIs) is that their behavior is tightly integrated with the context of their usage. As users invoke sequences of *events* (*e.g.*, *ClickOnCancelButton*, *TypeInText*) on GUI *widgets* (*e.g.*, `CancelButton`, `TextBox`), the underlying software responds (typically via the execution of an *event handler e.g.*, an ActionListener) in one of several ways. This may include a change to the software state, which may impact the execution of subsequent events. Hence, the *context*, established by the sequence of preceding events, in which an event executes may have an impact on *how* it executes. As Mathur notes [1], there is a close connection between events in a software system and its states. This context-sensitive and state-based execution behavior of GUI events creates problems for testing; each event needs to be tested in multiple contexts. Current model-based GUI testing techniques either test only a subset of event sequences by restricting the sequence to length two or length three [2], [3] or use a random method [4], rather than a systematic one, to test longer sequences [2], meaning that they can only consider a limited context.

The problem of event context and software testing is also relevant to the general class of event-driven software (EDS) [5] (sometimes termed as reactive software [6]–[8]), of which GUIs are a sub-class, as well as to any testing technique that generates sequence-based test cases, such as in state-based testing [9], [10]. In this paper we restrict our study of event context to GUIs, but believe that our techniques may be relevant to these other domains.

We use the term *GUI testing* to mean that a GUI-based software application is tested solely by performing sequences of events on GUI widgets; and the correctness of the software is determined by examining only the state of the GUI widgets. Although this type of testing interacts only with the GUI interface, the types of faults uncovered are varied. In recent work, Brooks *et al.* [11] characterized thousands of real faults detected via GUI testing and showed that a large proportion of faults detected are in the underlying business logic of the application, rather than in the GUI code itself.

The following example, obtained from our analysis of the Java open source program, `FreeMind` [12], helps to illustrate some real context related issues in a fielded GUI application. In this program there are four events we will call $\{e_1, e_2, e_3, e_4\}$ that correspond to $\{$*SetNodeToCloud*, *NewChildNode*, *NewParentNode*, *Undo*$\}$. The execution of either of the following 3-event test sequences $<e_1, e_2, e_3>$ or $<e_2, e_1, e_3>$ throws an *ArrayIndexOutOfBounds* exception that should have been found and fixed during testing. There are several interesting points to note. First, the combination $<e_2, e_3>$ triggers the fault only when e_1 provides context (*i.e.*, a cloud node) for this to happen. Hence, the *order* of the sequence of events is important; *i.e.*, if we test $<e_2, e_3, e_1>$, the fault is not detected.

Second, there is a difference between whether or not the events are tested consecutively. While $<e_1, e_2, e_3>$ triggers the fault, inserting e_4 at strategic points in the sequence, *e.g.*, $<e_1, e_2, e_4, e_3>$ causes the fault to go undetected. In this example, having event e_4 interrupt this sequence masks the fault; however, sometimes such an insertion may cause a previously undetected fault to be exposed. Suppose we had used shorter (length two) sequences for testing, (sometimes called *smoke tests* [3]). If we test $<e_2, e_3>$ the fault will be missed. But if we test this sub-sequence within a sequence of

- *X. Yuan has completed her Ph.D. and is currently a* Software Engineer in Test *at Google Kirkland.*
  E-mail: xyuan@cs.umd.edu
- *M. B. Cohen is with the Department of Computer Science and Engineering, University of Nebraska - Lincoln, Lincoln, NE USA.*
  E-mail: myra@cse.unl.edu
- *A. M Memon is with the Department of Computer Science, University of Maryland, College Park, MD 20742.*
  E-mail: atif@cs.umd.edu

events greater than length two, then we still have the possibility of detecting this fault because $<e_2, e_1, e_3>$ contains the added context of e_1.

Finally the absolute position of the event within the sequence affects fault detection. If the event sequence $<e_2, e_3>$ begins the test case (as the first two events of this sequence) we have no chance of detecting it. However, if it appears somewhere later, then a sequence such as $<e_1, \ldots, e_2, e_3>$ may detect this fault as well (unless "..." contains e_4 immediately after e_1, with no subsequent e_1). Similar test sequence specific faults have been highlighted elsewhere [2].

One method of modeling a GUI for testing creates a representation of events within windows (or components) called an *event-flow-graph* (EFG). Much like how a control-flow graph (CFG) encodes all possible execution paths in a program, an EFG represents *all possible sequences of events* that can be executed on the GUI. Coverage criteria based on the EFG have provided a notion of coverage of a GUI's event space for functional correctness [13]. However, these criteria cover sequences of events bounded by a specific length, which are essentially sub-paths through an EFG; they have the flavor of *path coverage* [14] in the traditional CFG sense. It is usually only possible to satisfy these criteria for *length two event sequence coverage* because the number of sub-paths grows exponentially with length. Hence, these criteria, as is the case with CFG path coverage, are useful from a theoretic perspective; they have limited practical significance.

Since there are a large number of events that do not interact with the business logic of the GUI application, such as those responsible for opening and closing windows, a refinement of the EFG was developed called an *event-interaction graph* (EIG) [2]. In an EIG events are modeled that do not pertain to the structural nature of the GUI (opening, closing windows, etc.) but that, instead, interact with the underlying application, called system interaction events [2]. We refer to testing only these events as *GUI interaction testing*.

These observations motivate a more complete examination of context-informed interaction testing in GUIs. To incorporate context we must first address the following limitations of the current techniques.

1) We lack GUI modeling methods that abstract the system interaction events in such a way that we can capture context in long event sequences.
2) We lack a systematic exploration of the impact of context-aware GUI interaction testing on fault detection.
3) We lack test adequacy criteria that sufficiently capture this new model of event sequences and that consider (i) event positions within test cases, (ii) whether certain events are consecutive or not, and (iii) test case length.

In a recent short paper [15], we explored ideas from *combinatorial interaction testing* (or CIT) [16], to study a new automated technique to generate long test cases for GUIs systematically sampled at a particular coverage strength, where a higher *strength* indicates that we are systematically testing more unique combinations of events. To facilitate this we developed a new abstraction of GUI system-interaction events. A preliminary feasibility study on one application showed that the CIT based technique was able to detect previously-undetected faults at a reasonable cost.

In this paper we explore these ideas more thoroughly. We present a family of context-aware GUI interaction testing criteria that use abstract event combinations, consider sequence length, and all possible positions of events within each sequence. We begin with a new model for events, that is an abstraction of the EFG, a *system interaction event set* (SIES). We then use key concepts from CIT [16], [17] to help describe our new criteria. Our motivation for using CIT as a starting point is that the coverage of a set of sequence-based test cases can be described and computed in terms of the 2-, 3-, 4-, or t-way relationships that are maintained between GUI events in all possible combinations of t-locations in the sequences. Our new GUI model, SIES, enables us to generate test cases using *covering arrays* [17] (see Section 2). We note, however, that the strict definition of CIT may be unnecessary for GUI testing and explore variations that are more cost effective, but that provide less coverage. We define a family of criteria with varying degrees of GUI interaction coverage.

We then embark on the first comprehensive study of event interaction coverage that considers context of events within test sequences. We present a large case study on eight well-studied applications to evaluate the effectiveness of this criteria on fault detection and test suite size.

The results of our study show that by increasing event combination strength and controlling starting and ending positions of events, our test cases are able to detect a large number of faults, not detected by exhaustive test suites of short tests. This increase is directly reflected in increased percentages of event strength coverage in our new criteria. Moreover, although the stronger of our new criteria require larger suites than the weaker criteria, these suites also detect additional faults. The specific contributions of this work include:

- New coverage criteria that consider event combination strength, sequence length, and all possible starting and ending positions for each event.
- An abstraction of the stateful GUI domain that allows us to recast the GUI test-case generation problem as one of combinatorial interaction testing.
- Evaluation of the new criteria via a large case study.

The next section provides related work on test sequences, some background on GUI testing, and an overview of combinatorial interaction testing. Section 3 describes the new CIT based adequacy criteria. Sections 4 through 6 present the design of the case study, its results, and limitations. Finally, Section 7 concludes with a discussion of future work.

2 Background and Related Work

A primary problem for GUI testing is that the length of the event sequence invoked by the user is often unbounded. There are an enormous number of possible permutations of these events which in turn means the context for testing is very large; testing all possible sequences a user may invoke is not possible. Instead, current GUI testing attempts to drive the software into different states by generating sequences that represent a sample of the entire state space.

Suppose, for example, that a user can invoke any of the following events on a drawing canvas in any order: $\{$*copy*, *paste*, *resize*, *rotate90*, *color*, *erase*$\}$. The sequence $<$*rotate90, color, copy, paste*$>$ may behave differently than the sequence $<$*rotate90, color, paste, copy*$>$ because execution of the event handler code for *copy* and *paste* may differ, *e.g.*, different values of variables may be read/written, different branches or paths may be executed. This relatively small set of events leads to 36 unique length-two sequences, over 7,500 unique length-five sequences, and more than 60 million unique length-ten sequences. During the execution of these sequences, the software may, in principle, transition through millions of different states.

Recent research has demonstrated that (1) GUI events interact in complex ways; a GUI's response to an event may vary depending on the context established by preceding events and their execution order [2], (2) GUI-event coverage is statistically related to the likelihood of detecting certain kinds of faults [18], (3) long test sequences are able to detect faults missed by short ones, even when the latter are systematically generated [19], and (4) events that interact directly with the underlying program business logic, as opposed to opening/closing menus/windows, are more likely to trigger faults [2]. This suggests that we need systematic coverage criteria for GUI testing that considers these issues.

There has been no prior work (other than in [15]) defining coverage criteria for GUI test sequences based on combinatorial interaction testing. However, several researchers have developed and studied criteria for test sequences in other domains, although context has not been their explicit focus in terms of event permutations and positions. Daniels *et al.* [20] define and compare a number of coverage criteria, for object-oriented software, based on method sequencing constraints for a class. The constraints impose restrictions on method behaviors and are derived from specifications of a class. Their goal is to execute sequences of instance methods that are obtained from the sequencing constraints and evaluate their results for correctness. Similarly, Farooq *et al.* [21] develop new coverage criteria based on colored Petri net models and used them for automatic generation of test sequences. They convert UML 2.0 activity diagrams, which are a behavioral type of UML diagram, into a colored Petri net. They define two types of structural coverage criteria for activity-diagram based models, namely sequential and concurrent coverage.

Several other researchers have relied on existing conventional criteria for test sequences. For example, Inkumsah *et al.* [22] use branch coverage to evaluate test cases, which are method sequences for object-oriented programs. Similarly, Gallagher and Offutt [23] use classical graph coverage criteria on data flow graphs for integration testing of object-oriented software that uses components that are developed by different vendors, in different languages, where the implementation sources are not all available. Gargantini *et al.* [24] use similar graph criteria on abstract-state machines to evaluate the adequacy of test cases generated from a model checker.

The work presented in this paper is unique in that it builds upon the foundation laid by combinatorial interaction testing [16], and applies this to GUI testing. This section first discusses prior work on GUI testing and then casts prior work on combinatorial interaction testing in GUI terms.

2.1 GUI Testing

A large body of research on software testing for GUIs exists [13], [25]–[31] and many GUI testing techniques have been proposed; some have been implemented as tools and adopted by practitioners. All of these techniques automate some aspect(s) of GUI testing including model creation (for model-based testing), test-case generation, test oracle creation, test execution, and regression testing. Although the nature and type of test cases may vary with different techniques, all of them explore the GUI's state space via sequences of GUI events.

Semi-automated unit testing tools such as *JFCUnit*, *Abbot*, *Pounder* and *Jemmy Module* [32] are used to manually create unit GUI test cases, which are then automatically executed. Assertions are inserted in the test cases to determine whether the classes/methods in the unit under test function correctly. More advanced tools called capture/replay tools "capture" a user session as a test case, which can later be "replayed" automatically on the GUI [33]. Again, test creation is manual and the tools facilitate only the execution of test cases. The part of the GUI state space explored by these test cases depends largely on the experience and knowledge of the testers and the quality of the user sessions.

Model-based techniques have been used to automate certain aspects of GUI testing. For example, manually created *state machine* models [25], [27] have been used to generate test cases. The nature and fault-detection effectiveness of generated test cases depend largely on the definition of "GUI states." Other work on GUI testing has focused on *graph* models to minimize manual work. The most successful graph models that have been used for GUI test-case generation include EFGs and EIGs [3]. The nodes in these graphs represent GUI events; edges represent different types of relationships between pairs of events.

An EFG models all possible event sequences that may be executed on a GUI. It is a directed graph that contains nodes (one for each event in the GUI) and edges that represent a relationship between events. An edge from node n_x to node n_y means that the event represented by n_y may be performed *immediately after* the event represented by n_x. This relationship is called `follows`. Note that a state-machine model that is equivalent to this graph can also be constructed – the state would capture the possible events that can be executed on the GUI at any instant; transitions cause state changes whenever the number and type of available events change. The EFG is represented by two sets: (1) a set of nodes **N** representing events in the GUI and (2) a set **E** of ordered pairs (e_x, e_y), where $\{e_x, e_y\} \subseteq \mathbf{N}$, representing the directed edges in the EFG; $(e_x, e_y) \in \mathbf{E}$ iff e_y `follows` e_x. An important property of a GUI's EFG is that it can be constructed semi-automatically using a reverse engineering technique called *GUI Ripping* [3]. A *GUI Ripper* automatically traverses a GUI under test and extracts the hierarchical structure of the GUI and events that may be performed on the GUI. The result of this process is the EFG.

EIG nodes, on the other hand, do not represent events to open or close menus, or open windows. The result is a more compact, and hence more efficient, GUI model. An EFG can be automatically transformed into an EIG by using graph-rewriting rules (details presented in [2]).

Figure 1 presents a GUI that consists of four events, *Cut, Copy, Paste*, and *Edit*. Figure 1(b) shows the GUI's EFG; the four nodes represent the four events; the edges represent the `follows` relationships. For example, in this EFG, the event *Copy* `follows` *Edit*, represented by a directed edge from the node labeled *Edit* to *Copy*.

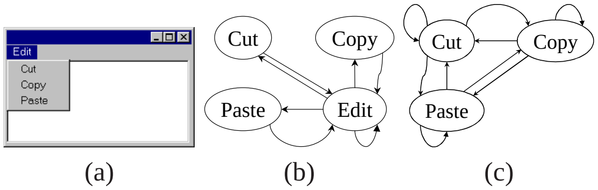


Fig. 1. (a) A Simple GUI, (b) its EFG, and (c) EIG.

Figure 1(c) shows the corresponding EIG. Note that the EIG does not contain the *Edit* event. In fact, the graph-rewriting rule used to obtain this EIG was to (1) delete *Edit* because it is a menu-open event, (2) for all remaining events e_x replace each edge $(e_x, Edit)$ with edge (e_x, e_y) for each occurrence of edge $(Edit, e_y)$, and (3) for all e_y, delete all edges $(Edit, e_y)$ and store the mapping "$e_y \rightarrow (Edit, e_y)$" for use during test execution. The GUI's EIG is fully connected with three nodes representing the three events.

The basic motivation of using a graph model to represent a GUI is that graph-traversal algorithms (with well-known run-time complexities) may be used to "walk" the graph, enumerating the events along the visited nodes, thereby generating test cases. A technique to generate test cases, each corresponding to an EIG edge has been developed; these test cases are called *smoke tests* [3]. Two examples of such length two smoke test cases for our example of Figure 1(c) are $<Copy, Cut>$ and $<Cut, Paste>$. There are a total of nine such tests – one for each EIG edge. Because EIG nodes do not represent events to open/close menus or open windows, other events (in this case *Edit*) needed to reach the EIG events are automatically generated at execution time using the mappings $\{Cut \rightarrow (Edit, Cut)$, $Paste \rightarrow (Edit, Paste)$, $Copy \rightarrow (Edit, Copy)\}$ stored earlier, yielding an executable test case [3]. The two test cases will "expand" to $<Edit, Copy, Edit, Cut>$ and $<Edit, Cut, Edit, Paste>$.

Based on these graph models, a class of coverage criteria called *event-based criteria* has been defined [13]. These criteria use events and event sequences to specify a measure of GUI test adequacy. A *GUI component* is defined as the basic unit of testing. The GUI is represented by its components and their interactions. Two types of criteria are defined: (1) intra-component criteria for events within a component and (2) inter-component criteria for events across components. However, these criteria did not account for context, sequence length, and position of events in a test case.

In more recent work [15], we used covering arrays to generate long event sequences. Although we did not have a notion of test adequacy, our test cases were useful – a feasibility study on one subject application showed that the new technique was able to detect faults that were previously undetected. Our current work formalizes the notion of using combinatorial interaction testing by defining adequacy criteria that capture context.

Events:{*ClearCanvas, DrawCircle, Refresh*}

CoveringArray: ***CA(9;2,4,3)***			
ClearCanvas	*ClearCanvas*	*ClearCanvas*	*ClearCanvas*
ClearCanvas	*Refresh*	*Refresh*	*DrawCircle*
ClearCanvas	*DrawCircle*	*DrawCircle*	*Refresh*
DrawCircle	*ClearCanvas*	*Refresh*	*Refresh*
DrawCircle	*DrawCircle*	*ClearCanvas*	*DrawCircle*
DrawCircle	*Refresh*	*DrawCircle*	*ClearCanvas*
Refresh	*ClearCanvas*	*DrawCircle*	*DrawCircle*
Refresh	*Refresh*	*ClearCanvas*	*Refresh*
Refresh	*DrawCircle*	*Refresh*	*ClearCanvas*

Smoke Tests
1.<ClearCanvas, ClearCanvas> *2.<ClearCanvas, DrawCircle>* *3.<ClearCanvas, Refresh>* *4.<DrawCircle, DrawCircle>* *5.<DrawCircle, Refresh>* *6.<DrawCircle, ClearCanvas>* *7.<Refresh, Refresh>* *8.<Refresh, ClearCanvas>* *9.<Refresh, DrawCircle>*

Fig. 2. Covering Array and Smoke Tests

2.2 Combinatorial Interaction Testing

The basis for combinatorial interaction testing is a *covering array*. A covering array (written as $CA(N; t, k, v)$) is an $N \times k$ array on v symbols with the property that every $N \times t$ sub-array contains all ordered subsets of size t of the v symbols *at least* once [17]. In other words, any subset of t-columns of this array will contain all t-combinations of the symbols. We use this definition of a covering array to define the GUI event sequences.[1] Suppose we want to test sequences of length four and each location in this sequence can contain exactly one of three events (*ClearCanvas, DrawCircle, Refresh*) as is shown in Figure 2. Testing all combinations of these sequences requires 81 test cases. We can instead sample this system, including all sequences of shorter size, perhaps two. We model this sequence as a $CA(N; 2, 4, 3)$ (top portion of Figure 2). The *strength* of our sample is determined by t. For instance we set t=2 in the example and include all pairs of events between all four locations. If we examine any two columns of the covering array, we will find all nine combinations of event sequences at least once. In this example there are 54 event sequences of length two which consider the sequence location. This can be compared with testing *only* the nine event sequences which would be used in our prior generation technique for smoke tests (see bottom portion of Figure 2).

[1] A more general definition for a covering array exists, that does not assume a single v, but instead allows each location in the array to have a different number of symbols. This type of array is not necessary for our problem, since we will always have the same number of events in each of the k positions.

The number of test cases required for the t-way property, is N. In our example, we generate a $CA(9;2,4,3)$, *i.e.*, a 9×4 array on the 3 events with the property that every 9×2 sub-array contains all ordered subsets of size 2 of the 3 events *at least* once. Since the primary cost of running the test case is the setup cost, we cover many more event sequences for almost the same cost as our smoke tests. In general we cannot guarantee that the size of N will be the same as the shorter sequence, but it will grow logarithmically in k rather than exponentially as does number of all possible sequences of length k [16].

Covering arrays have been used extensively to test input parameters of programs [16], [34], [35] as well as to test system configurations [36]–[38]. Other uses of covering array sampling have been suggested, such as testing software product line families [39] and databases [40]. A special type of a covering array, (an orthogonal array) developed from Latin squares, has been previously used to define GUI tests by White [31]; however this work used covering arrays in a stateless manner, to define subsets of the input parameter combinations. Bryce *et al.* used covering arrays to test a flight guidance system also modeled with state variables [41]; however, only event sequences of length one were considered. In this paper, we use covering arrays to sample long event sequences, where events must consider state (determined by location in sequence and all prior events).

3 Event Coverage Test Adequacy

This section presents a family of event coverage adequacy criteria created to capture our notion of interaction coverage. The strongest criterion (described last) is derived directly from a covering array, while the other criteria are relaxations of this sampling technique. They are meant to capture specific types of context manifested by consecutive and interrupting events.

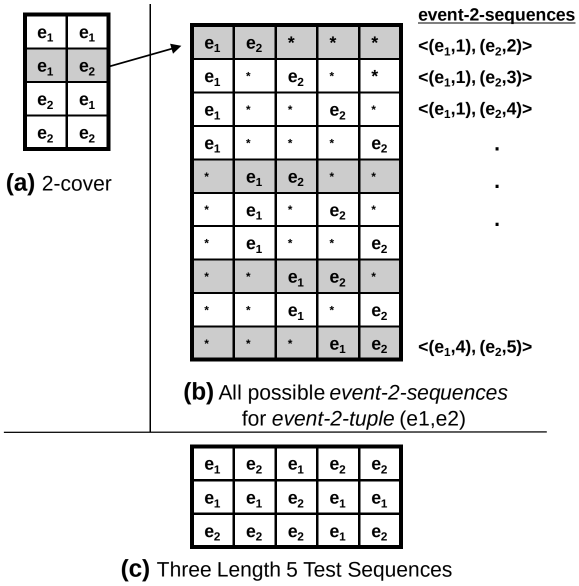
We begin by defining event-tuples and event-positions in a sequence. Assume we have a set of events E, and each event can occupy any location p in the sequence S of length k. Our first definition does not assume a specific position (or context) within a sequence of the events, but rather defines a combination of events that occur in order somewhere within the sequence.

Definition: An *event-t-tuple* $(e_i, e_j, \ldots, e_t)$ is an ordered tuple of size t of events from E. A set of events E gives rise to $|E|^t$ *event-t-tuples*, *i.e.*, all possible permutations of events.

The example shown in Figure 3 labels two events e_1 and e_2 (other events are not labeled). There are four possible *event-2-tuples* for these two events – (e_1, e_1), (e_1, e_2), (e_2, e_1), and (e_2, e_2) shown in Figure 3(a). In a sequence of length 5, shown in Figure 3(b), these *event-2-tuples* can occur with and without other events between them and in any location in the sequence.

To account for context, we need to associate positions within a sequence to specific events. The next few definitions allow for context.

Definition: An *event-position* in a length-k sequence S is an ordered pair (e, p), where event $e \in E$ is at position p $(1 \leq p \leq k)$.



e_1	e_1
e_1	e_2
e_2	e_1
e_2	e_2

(a) 2-cover

					event-2-sequences
e_1	e_2	*	*	*	<(e_1,1), (e_2,2)>
e_1	*	e_2	*	*	<(e_1,1), (e_2,3)>
e_1	*	*	e_2	*	<(e_1,1), (e_2,4)>
e_1	*	*	*	e_2	.
*	e_1	e_2	*	*	.
*	e_1	*	e_2	*	.
*	e_1	*	*	e_2	
*	*	e_1	e_2	*	
*	*	e_1	*	e_2	
*	*	*	e_1	e_2	<(e_1,4), (e_2,5)>

(b) All possible *event-2-sequences* for *event-2-tuple* (e1,e2)

e_1	e_2	e_1	e_2	e_2
e_1	e_1	e_2	e_1	e_1
e_2	e_2	e_2	e_1	e_2

(c) Three Length 5 Test Sequences

Fig. 3. Event coverage and Event-position coverage

In the first row of the table seen in Figure 3(b), we see event e_1 in position 1 of the sequence; therefore this is event-position $(e_1, 1)$. Similarly, for e_2 in the same sequence, the event position is $(e_2, 2)$.

Given a sequence S of length k, we now extend the event-position concept to a vector.

Definition: An *event-t-sequence* is a vector of *event-positions* of length t, $<(e_i, p_1), (e_j, p_2), ..., (e_n, p_t)>$, where $k \geq t$, $1 \leq p_x \leq k$ for all x, $1 \leq x \leq t$, $p_1 < p_2 < ... < p_t$, and $e_i, e_j, ...e_n \in E$.

In Figure 3(b), $<(e_1, 1), (e_2, 2)>$ in the first row and $<(e_1, 1), (e_2, 3)>$ in the second row are both *event-2-sequences*. Note that the definition includes the same events in different positions; hence it is perfectly reasonable to have $<(e_1, 1), (e_1, 2)>$ in a sequence if e_1 appears in positions 1 and 2. A single length k-sequence, S, with $k \geq t$ has $\binom{k}{t}$ *event-t-sequences* within it.

As the `FreeMind` example in Section 1 illustrates, for GUI fault detection, it is important to distinguish between event sequences in which certain events are *required* to appear consecutively versus allowing other events to appear in between, thereby establishing a different context of execution. We now allow for this distinction, leading to two different concepts of test adequacy.

Definition: An *event-consecutive-t-sequence* is an *event-t-sequence* $<(e_i, p_1), (e_j, p_2), ..., (e_k, p_t)>$ such that $p_x = p_{x-1} + 1$, for all $1 < x \leq t$.

That is, the events in an *event-consecutive-t-sequence* must be in adjacent positions. In 3(b), we have shown the expansion of all possible positions for the *event-2-tuple* (e_1, e_2) in a length 5 sequence. The *event-consecutive-2-sequences* are shown in the highlighted rows.

We can use these definitions to develop our first notion of event coverage adequacy. The coverage is tied to both the event combination strength t, as well as the length of the sequence k. Context comes from the notion of position in the sequence.

Definition: A test suite is t-*cover* adequate if it executes all possible *event-t-tuples* (all permutations of events) in the form of an *event-consecutive-t-sequence* at least once.

Examining the set of three length 5 test sequences in Figure 3(c), we see that all four *event-2-tuples* appear in the form of an *event-consecutive-2-sequence* at least once. For example, the first test sequence covers (e_1, e_2), (e_2, e_1) and (e_2, e_2); the second covers (e_1, e_1). This set of test sequences has 100% 2-*cover* adequacy. If we generate only length-2 sequences (which in fact are our smoke tests), then for 100% 2-*cover* adequacy, we need the four sequences shown in Figure 3(a). Whenever t==k, a t-*cover* adequate test suite is equivalent to an exhaustive enumeration of sequences of length k. In general, longer, carefully chosen sequences should allow us to achieve better coverage using fewer sequences.

Going back to our `FreeMind` example of Section 1, a 3-*cover* adequate suite for this application would contain all possible *event-3-tuples* in the form of an *event-consecutive-3-sequence*, including the exception-causing sequences $<e_1, e_2, e_3>$ and $<e_3, e_2, e_1>$.

We now consider event sequences that are not consecutive. Given a sequence S of length k, where $k > t$, if we have at least one *event-t-sequence* contained within it that is non-consecutive (*i.e.*, there is at least one *event-position* interrupting this sequence), we call this an *event-non-consecutive-t-sequence*.

Definition: An *event-non-consecutive-t-sequence* is an *event-t-sequence* $<(e_i, p_1), (e_j, p_2), ...(e_n, p_t)>$, where $p_1 < p_2 < ... < p_t$, such that at least one interval $(p_2 - p_1), ..., (p_t - p_{t-1})$ is greater than 1.

In Figure 3(b), the non-shaded rows represent *event-non-consecutive-2-sequences* for the *event-2-tuple* (e_1, e_2).

This brings us to our next adequacy coverage metric.

Definition: A test suite is t^+-*cover* adequate if it executes all possible *event-t-tuples* in the form of an *event-non-consecutive-t-sequence* at least once. Adequacy is zero when t==k.

In Figure 3(b), there are 6 possible *event-non-consecutive-2-sequences* to choose from for each of the *event-2-tuples*. To satisfy a 2^+-*cover*, we need to select only one of each. For instance in rows 2, 3 and 4, we have a single *event-2-tuple* represented by different *event-non-consecutive-2-sequences*. Only one of these is needed to satisfy coverage. In Figure 3(c), the three test sequences cover all *event-2-tuples* in an *event-non-consecutive-2-sequence* at least once; therefore it has 100% 2^+-*cover* adequacy.

We can combine these two coverage metrics to get a third notion of adequacy.

Definition: A test suite is t^*-*cover* adequate if it is both t-*cover* adequate and t^+- *cover* adequate for a common set of events E.

A 2^*-*cover* for Figure 3(a) requires at least 8 *event-t-sequences* (4 *event-consecutive-2-sequences* and 4 *event-non-consecutive-2-sequences*). We have 100% 2-*cover* adequacy, but 0% 2^+-*cover* adequacy (recall that by definition this is 0 when t==k). Therefore we have only 50% 2^*-*cover* adequacy. However, Figure 3(c) has both 100% 2-*cover* and 2^+-*cover* adequacy as well as 100% 2^*-*cover* adequacy.

If we want to consider context more completely for testing, then the length of S when $k > t$, dictates a stronger set of testing contexts; *i.e.*, testing from all possible starting states. There are $\binom{k}{t} \times |E|^t$ possible *event-t-sequences* which means that the same set of *event-t-tuples* can be tested from many different positions. This is not captured in our adequacy so far. To define this criteria we use a covering array from traditional CIT and extend it to the notion of sequences defined above.

Definition: A test suite with test cases of length k is *t-k-covering array* adequate when $t < k$ and it contains all possible *event-t-sequences* at least once. Covering array adequacy is not defined when t==k since this is an exhaustive enumeration of all possible event permutations.[2]

A *2-5-covering array* for the events in Figure 3(a) requires that we include all of the possible combinations of *event-2-sequences*. There are 40 combinations that must be covered in a length 5 sequence for this set of 2 events.

If we examine the test sequences in Figure 3(c), although we have 100% adequacy for a 2^*-*cover*, the *2-5-covering array* adequacy is only 70% (28 of the 40 *event-2-sequences* are covered). For instance, the *event-2-sequence* $<(e_1, 2), (e_2, 4)>$ seen in row six of Figure 3(b), is missing from the set of test sequences.

So far we have measured adequacy for t=2, but we can easily extend this to t=3,4,..., etc. The 3-*cover* adequacy for Figure 3(c) is 87.5%. We are missing an *event-consecutive-3-sequence* for the *event-3-tuple* (e_1, e_1, e_1). The 3^+-*cover* is also missing an *event-non-consecutive-3-sequence* for the *event-3-tuple* (e_2, e_1, e_1). Together they provide 87.5% adequacy for the 3^*-*cover*. The *3-5-covering array* adequacy is much lower. Only 30 *event-3-sequences* are covered (out of 80) for an adequacy of 37.5%.

Our adequacy criteria have a clear subsumption relationship when $k > 2$. The *t-k-covering array* subsumes all other adequacy criteria, while the t^*-*cover* subsumes both t^+-*cover* and t-*cover* adequacy.

4 Case Study

We now describe a case study to systematically explore the effectiveness of our new adequacy criteria on event context. The overall research question that we aim to answer is: "*Is there a correlation between the defined coverage-adequate test criteria and fault detection?*" To answer this question we begin with a characterization of test suites using the strongest coverage criteria defined, t-k-*covering array* adequate test coverage. Our characterization first examines fault detection and then quantifies the coverage of subsumed adequacy within these test suites. We then examine how specific test suites developed for each of the adequacy criteria perform with respect to fault detection.

4.1 Study Setup

We selected two different sets of subjects for our study. The first set consists of four applications: `TerpPaint`, `TerpPresent`, `TerpSpreadSheet`, and `TerpWord` of

[2] This differs from the traditional definition of a covering array [17] where $t \leq k$.

the TerpOffice suite developed at the University of Maryland.[3] We have used TerpOffice in numerous experiments before, and are very familiar with its code and functionality, which is very important as Step 2 of the study procedure will show. To minimize threats to external validity, our second set consists of four open source GUI-based applications (CrosswordSage 0.3.5, FreeMind 0.8.0, GanttProject 2.0.1, JMSN 0.9.9b2) downloaded from SourceForge. These applications have also been used in our previous experiments [42]; details of why they were chosen have been presented therein.

For the TerpOffice applications, a GUI fault is defined as a mismatch, detected by a test oracle, between an "ideal" (or expected) and actual GUI state. Hence, to detect faults, a description of ideal GUI execution state is needed. We develop the test oracle from a "golden" version of the subject application and use the oracle to test other *fault-seeded versions* of the application. An automated tool executes each event in each test case on the golden version, and captures the GUI state (widgets, properties, and values) automatically by using the Java Swing API. The state information is then converted to a sequence of `assertEquals(X, Y)` statements, where `X` is the extracted value of a widget's property. `Y` is a placeholder that is instantiated with the corresponding value extracted from the fault-seeded version. The method `assertEquals()` returns `TRUE` if its two parameters are equal, otherwise `FALSE`. The test cases are also executed on each fault-seeded version (one fault per version). The results of the `assertEquals()` are recorded. If, after an event e in a test case t executes on fault seeded version F_i, even one `assertEquals()` method returns `FALSE`, then t is said to have "detected the fault F_i." The number of seeded faults relevant to this study seeded in TerpPaint, TerpPresent, TerpSpreadSheet, and TerpWord, are 116, 126, 114, and 71, respectively.

In the SourceForge application set we rely on *naturally occurring* faults. Here, the application is said to have passed a test case if it did not *crash* (terminate unexpectedly or throw an uncaught exception) during the test case's execution; otherwise it failed. Such crashes may be detected automatically by the script used to execute the test cases. Due to their popularity, these applications have undergone quality assurance before release. To further eliminate "obvious" bugs, we used a static analysis tool called *FindBugs* [43] on these applications; after the study, we verified that none of our reported bugs were detected by FindBugs.

We generate and execute smoke tests using the EIG-based algorithm for all subjects; the faults detected by these test cases are labeled "smoke faults" and no longer used in this study (they are not included in the seeded fault numbers mentioned above).

4.2 Study Procedure

The study is conducted in six steps on each subject application independently, as described next. The most complex part of this study is the completion of the first three steps through test-case generation. To achieve our new coverage criteria, we need to control permutations of events and their positions; covering arrays provide the strongest coverage criteria. However, they also provide some unique challenges that need careful modeling and preparation.

When combining events, one must consider that GUI events have strict structural constraints (*e.g.*, the event *PrinterProperties* in the Print window can be executed only after the Print window is open) and complex GUI-state-based dependencies (*e.g.*, execution of one event results in a GUI state in which another event is enabled/disabled); one cannot simply concatenate different events together to obtain a single executable test case – certain events may be unavailable or disabled. Our study procedure explicitly models and adds state-based constraints that reduce the need for structural ordering relationships between GUI events. The first step in the study models and creates the system interaction event set (SIES) to be used as the basis for the covering array algorithm. The second step generates the test cases for our study. The third step takes the output of the covering arrays and converts them into executable tests. The fourth and fifth step run our tests, detect faults and capture coverage while the last step analyzes our results for the various coverage criteria developed in Section 3. Details for each step are described next.

Step 1. Prepare the GUI model for Covering Arrays: In this step we use our reverse engineering tool *GUI Ripper* to create the event-interaction graph (EIG) model. Details of the *GUI Ripper* have been described in earlier reported work [3]. Here, it is sufficient to know that the *Ripper* automatically traverses the GUI structure of the program under study; the output is the EIG. The most important property of EIGs for this work is that the EIG may not contain all the events in the GUI; some events are abstracted away; we will revisit the impact of this property in Step 4.

We next group the events by functionality. The events within each group constitute the events for a single model that are used to generate test sequences. Events that are not contained within the same group will not be tested together. However, one event may be a part of multiple groups. This part of our process is currently done manually. Domain knowledge is required to determine which events are likely to be included in similar functionality. Future work is to automate this process through the use of historical data on similar domains. The output is a model that lists the specific event groups as well as the number of events per group. The groups and their number of events (#Events(v)) are shown for each of our applications in Table 1; note that because groups may be overlapping, the "Total" column is not the sum of events in the groups shown – the Total column shows the number of *unique* events in the groups.[4]

Once the event graphs and groups have been identified, it is necessary to specify constraints on events such that the generated event sequences are executable. This is necessary because some events may not run without a set of prior set-

[3] Detailed specifications, requirements documents, source code CVS history, bug reports, and developers' names are available at `http://www.cs.umd.edu/users/atif/TerpOffice/`.

[4] Complete experimental results can be found at `http://www.cs.umd.edu/users/atif/tse09/`.

Groups	1	2	3	4	5	6	7	8	9	Total
TerpPaint Description	Tool Mgt.	Image Settings	Clipboard Ops.	Layer Manip.	File Ops.					
#Events(v)	27	35	11	11	6					88
TerpPresent Description	View	Format	Text	Shape	Content	Clipboard	Windows	Tools		
#Events(v)	14	20	31	13	14	11	7	10		102
TerpSpreadSheet Description	Format Cell	Cell Function	Graphs	Content	Table Format	Find/ Replace				
#Events(v)	14	12	4	12	8	5				46
TerpWord Description	Table	Document Style	Content	Ins. Image Properties	Font Settings	Clipboard	Window Style	Searching	Manage Plugins	
#Events(v)	14	32	16	7	12	8	3	11	3	92
CrosswordSage Description	Manage Crossword	Solve Word	Open and Save	Preference Settings						
#Events(v)	11	6	4	14						33
FreeMind Description	Map Ops.	Format	Edit Node	Clipboard Ops.	File Ops.	Node Ops.	Tools			
#Events(v)	11	18	16	10	23	17	10			99
GanttProject Description	Mgmt.	File Ops.	Format	Print Preview	Settings	Task Properties				
#Events(v)	14	14	13	12	7	25				85
JMSN Description	Logon	View	Tools	Settings	Report					
#Events(v)	4	15	18	18	8					63

TABLE 1
Event Grouping in Subject Applications

up events, or must occur only after another event has been fired. For instance, in `TerpSpreadSheet`, the *Undo* event requires that one of the events *Cut*, *Paste*, *PasteSpecial*, *Clear* occurs first; otherwise *Undo* remains disabled; we will use these constraints in Step 4. This ends the modeling stage of our test case generation.

Step 2. Generate Covering Arrays: We generate t-10-*covering arrays* for our test cases; i.e., strength t covering arrays with 10 columns. There are three inputs to this step of the process. The first is k which determines the length of our abstract sequences (*i.e.*, those that may need the insertion of other events to become executable). In this study, k=10 for all our subjects. This was chosen heuristically as the longest feasible size for overall success with our test harness. The second is the number of abstract events per location, v, with a list of the v abstract events that are to be tested. This comes from the previous step. The third is the strength of the desired covering array, t. Using these parameters, a covering array is generated using one of the known covering array generation algorithms; simulated annealing [17].

The covering array contains N rows consisting of abstract events. In this study, we generate covering arrays for consecutive successive strengths starting at t=2, until our arrays reach sizes larger than 20,000 test sequences. This number was chosen due to resource limitations. The sizes of these covering arrays and the number of test cases generated per group is shown in Section 5, Tables 4 and 5. The strengths of covering arrays in our subjects ranges from t=2 to t=8. (Two groups of `TerpWord` had only three events allowing us to increase t to 8.) The majority of our groups are not tested beyond t=4. These covering arrays, each row corresponding to an abstract event sequence, are then passed to the next phase for translation into executable tests.

Step 3. Generate Executable Tests: The abstract event sequences from Step 2 are expanded in this step to generate executable test cases. The expansion process is done in two stages. We will explain these stages via an example. Consider an actual abstract event sequence <*WriteInTable*, *ComputeAverage*, *WriteInTable*, *Undo*, *InputFunction*, *Count*, *InputFunction*, *ComputeSum*, *ComputeMin*, *ComputeSum*> from the 2-way covering array for Group 2 of `TerpSpreadSheet`. First, the constraints from Step 2 are used to increase the test case's chances of executing to completion, by ensuring that none of the events are disabled. In this example sequence, we know that *Undo* will remain disabled unless it is preceded with specific events; hence event *Copy* is inserted before *Undo*, resulting in the expanded sequence <*WriteInTable*, *ComputeAverage*, *WriteInTable*, *Copy*, *Undo*, *InputFunction*, *Count*, *InputFunction*, *ComputeSum*, *ComputeMin*, *ComputeSum*>. However, certain parts of the sequence are still not executable; for example, the subsequence <*WriteInTable*, *ComputeAverage*> cannot be executed; event *ComputeAverage* is not available in the GUI after event *WriteInTable* has been executed. Additional events are needed to "drive" the GUI so that *ComputeAverage* becomes available. Hence, in the second stage of expansion, some events that were abstracted away in Step 1, are now re-inserted to "reach" other events; these reaching events are obtained automatically [2]. The fully expanded sequence is <*WriteInTable*, *Function(Menu)*, *ComputeAverage*, *WriteInTable*, *Edit(Menu)*, *Copy*, *Edit(Menu)*, *Undo*, *InputFunction*, *Function(Menu)*, *Count*, *InputFunction*, *Function(Menu)*, *ComputeSum*, *Function(Menu)*, *ComputeMin*, *Function(Menu)*, *ComputeSum*>, where events *Function(Menu)* and *Edit(Menu)* correspond to "click-on-Function-menu" and "click-on-Edit-menu", respectively.

Step 4. Execute Test Cases: Each application is launched in a state in which most of the events are enabled – in all cases, this requires loading a previously saved application file. Because we need 1,257,619 test runs, we use a cluster of 40 machines to speed up execution; all are Pentium 4 2.8GHz Dual-CPUs with 1GB memory, running Red Hat Enterprise Linux version 2.6.9-11. The total time used is 9 days per machine. Data is collected via our automated oracle to determine test cases that detect faults; this forms the error report. (Later analysis of the error reports confirmed that the faults detected by the smoke tests were subsumed by the covering-array-based tests.)

Step 5. Compute Coverage: The input to this step is the set of test cases, and the last successfully executed location in the sequence. This data is analyzed against all of our coverage criteria. We calculate the adequacy up through the last executed sequence in a test case. Our denominator is the number of required t-tuples and their associated *event-positions* for the particular adequacy criterion being measured. The results of this computation are discussed in Section 5.

Step 6. Analyzing Test Adequate Suites: Our final step is to analyze our data for the various coverage criteria developed in Section 3. We use a greedy technique for this process. Since our t-10-*covering array* contains the highest possible coverage, we use these test cases as our starting point. For each group and each strength (t) that finds at least one fault, we find a subset of test cases from its corresponding covering array that will satisfy the maximum coverage for each of the adequacy criteria, one row at a time. We begin by selecting a test case (randomly from among the best when there is a tie) that gives us the highest coverage for the adequacy criteria desired. We then select the test case that has the largest increase in cumulative coverage. We continue adding test cases until we have reached the maximum coverage obtainable for that criterion in our covering array. For instance if we are interested in a 2-$cover$ adequate test suite and the original 2-10-*covering array* for that subject/group was 95% 2-$cover$ adequate, we select test cases until we reach 95% coverage. We note that we do not restrict the test cases to those which are length 10. Although most of the selected test cases are length 10, it is possible that ones which did not run to completion (i.e., have a shorter length) are selected by the algorithm, when this provides the largest increase in coverage. To obtain our t^*-*cover* suites we take a set union (eliminating duplicates) of the corresponding *t-cover* and t^+-*cover* test suites. This is an upper bound on the size of the suites, since we may be able to remove duplicate coverage, but we are guaranteed to have complete t^* coverage by doing this. To compute the fault detection of each coverage adequate test suite we analyze the fault matrices from the covering array test execution runs.

4.3 Practical Limitations of Study Procedure

As is the case for conventional software, where there are infeasible paths in programs, we may have infeasible sequences in our test cases. These infeasible sequences cause some of our test cases to not run to completion. Similar problems have been noted in other state-based testing domains [9]. However, we do not know which event sequences are infeasible *a priori*. The impact of this situation on test adequacy is expected to be similar to that of adequacy for statement or branch coverage in a program. We can only cover 100 percent of the *feasible* sequences. In this work we do not attempt to repair infeasible sequences, but leave that as a future avenue to pursue. The parts of the test cases that did not execute is recorded; we use this information to characterize our test cases in the next section.

In this study, we devised several domain-specific modeling steps to reduce the generation of those event sequences that execute infeasible paths. Many of these steps are done manually. First, we manually group events together by functionality. Second, we manually identify constraints on events such that the generated event sequences are likely to be executable. We consider this manual effort to be a serious limitation of our test-case generation techniques. We will pursue this limitation in future work.

4.4 Independent and Dependent Variables

For our study the independent variables are the coverage criteria described in the previous section, t-10-*covering array* adequate test suites, t-$cover$, t^+-$cover$ and t^*-$cover$ test suites. We use a single t-10-*covering array* for each subject/group/strength, but we generate five test suites for each of the other coverage-adequate test suites. We do this to reduce the likelihood that a random choice of test case impacts our results. The dependent variables are the covered percentage for each criterion defined, fault detection and test suite size.

4.5 Threats to Validity

We have tried to reduce threats to validity of our study but as with all experiments there are limitations to the results that can be inferred. We discuss the main threats here. The first threat that we identify is the threat to generalization of our results. Four of our subjects (the `TerpOffice` applications) were written by students in a University setting. We believe however that these are realistic GUI applications and they have been used in many other studies. To reduce this threat, we have selected four open source applications as well. Another threat is that we have run only a single covering array for each strength of each group, but we have generated each one independently of the others and believe that the general trends are still valid. Given the large resources required to run all of these tests suites we do not believe that we could have performed such a broad study and run multiple covering arrays at each strength as well.

The second threat we address is that of internal validity. All of our testing is performed in an automated fashion and we use many programs to analyze the resulting data. We have validated our programs and have hand analyzed subsets of the data to confirm the results obtained are accurate. We have cross checked our adequacy of suites using more than one program.

Finally it is possible that we have selected the wrong metrics (construct validity) for study, but we believe that fault detection, coverage and test suite size are important metrics that impact test suite effectiveness and are a reasonable starting point for evaluation.

5 Characterization of Generated Test Suites

To answer our research question we begin with a characterization of the covering array test suite developed in Step 2. In this section we quantify the fault detection of our covering array test suites which represents the maximum possible fault detection. We then quantify the various adequacy criteria within each test suite. In the next section we examine the effectiveness of the different adequacy criteria test suites developed in Step 6. We examine this with respect to both fault detection and the size of the test suites.

5.1 Overall Fault Detection

Since all subjects were tested prior to this study with smoke tests, any fault found in this study is *new* and considered undetectable by them. Table 2 shows the detailed fault data for each subject of `TerpOffice` for the highest adequacy criterion, t-10-*covering arrays*. In this table each row is a test suite labeled $TCA_{t=n}$, where TCA stands for the covering array test suite and t=n is the strength of the covering array for testing. The columns of this table are the different groups tested in each application. A dash in a cell means that we did not have a test suite at that strength. The totals shown for each test suite and each group represent the total *unique* faults found. The columns and rows cannot simply be added since there is overlap in fault detection for both the test suites and for the groups; some groups share events and faulty code.

Test Suite	Groups									Total
	1	2	3	4	5	6	7	8	9	
TerpPaint										
$TCA_{t=2}$	69	3	5	0	0	-	-	-	-	76
$TCA_{t=3}$	-	-	8	0	0	-	-	-	-	8
$TCA_{t=4}$	-	-	-	-	5	-	-	-	-	5
Total	69	3	8	0	5	-	-	-	-	82
TerpPresent										
$TCA_{t=2}$	1	1	0	0	10	44	0	94	-	105
$TCA_{t=3}$	5	5	-	0	10	50	0	114	-	119
$TCA_{t=4}$	-	-	-	-	-	-	0	-	-	0
Total	5	5	0	0	10	50	0	114	-	119
TerpSpreadSheet										
$TCA_{t=2}$	17	1	2	9	1	9	-	-	-	35
$TCA_{t=3}$	19	1	2	35	1	10	-	-	-	65
$TCA_{t=4}$	-	-	2	-	1	9	-	-	-	12
$TCA_{t=5}$	-	-	2	-	-	11	-	-	-	11
Total	19	1	2	35	1	11	-	-	-	66
TerpWord										
$TCA_{t=2}$	0	1	16	0	0	0	0	0	0	17
$TCA_{t=3}$	1	-	39	0	0	0	0	0	0	39
$TCA_{t=4}$	-	-	-	0	-	0	0	-	0	0
$TCA_{t=5}$	-	-	-	-	-	-	0	-	0	0
$TCA_{t=6}$	-	-	-	-	-	-	0	-	0	0
$TCA_{t=7}$	-	-	-	-	-	-	0	-	0	0
$TCA_{t=8}$	-	-	-	-	-	-	0	-	0	0
Total	1	1	39	0	0	0	0	0	0	40

TABLE 2
`TerpOffice` Covering Array
Test Suite Fault Detection

Test Suite	Groups							Total
	1	2	3	4	5	6	7	
CrosswordSage								
$TCA_{t=2}$	0	0	0	0	-	-	-	0
$TCA_{t=3}$	0	0	0	0	-	-	-	0
$TCA_{t=4}$	-	0	0	-	-	-	-	0
$TCA_{t=5}$	-	-	0	-	-	-	-	0
Total	0	0	0	0	-	-	-	0
FreeMind								
$TCA_{t=2}$	0	0	1	1	0	1	0	3
$TCA_{t=3}$	0	0	2	1	-	3	0	4
Total	0	0	2	1	0	3	0	4
GanttProject								
$TCA_{t=2}$	0	1	0	0	0	0	-	1
$TCA_{t=3}$	3	1	2	2	0	-	-	7
$TCA_{t=4}$	-	-	-	-	0	-	-	0
Total	3	1	2	2	0	0	-	7
JMSN								
$TCA_{t=2}$	0	0	0	0	0	-	-	0
$TCA_{t=3}$	0	0	0	0	0	-	-	0
$TCA_{t=4}$	0	-	-	-	0	-	-	0
$TCA_{t=5}$	0	-	-	-	-	-	-	0
Total	0	0	0	0	0	-	-	0

TABLE 3
`SourceForge` Covering Array
Test Suite Fault Detection

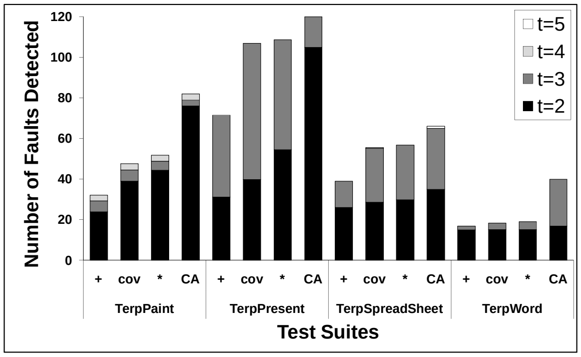


Fig. 4. Cumulative Fault Coverage: (`TerpOffice`)

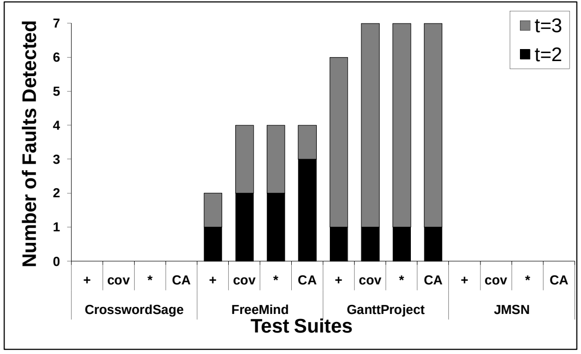
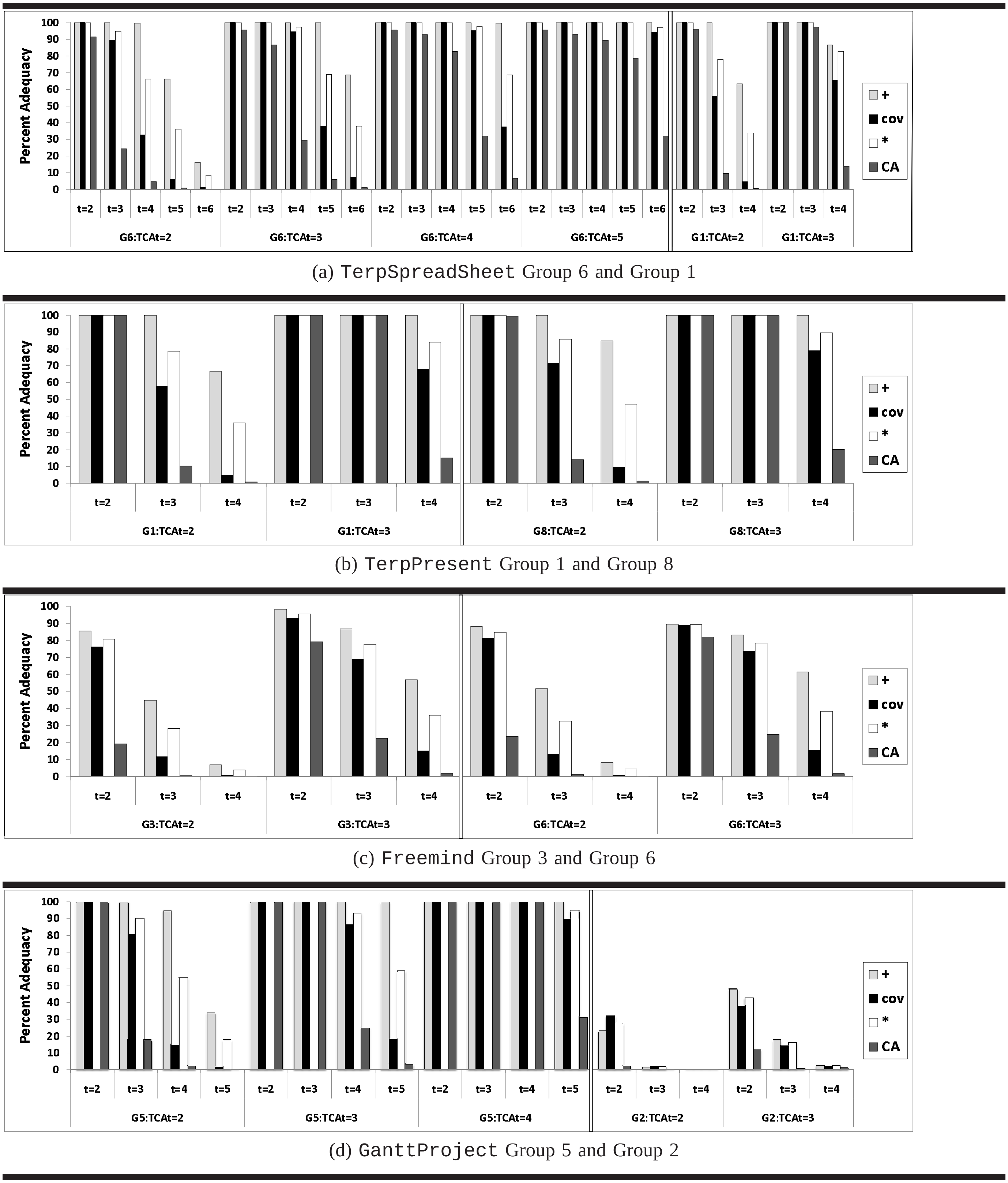


Fig. 5. Cumulative Fault Coverage: (`SourceForge`)

In the `TerpOffice` applications we detected a large number of faults that were previously undetected. In `TerpPaint` we found a total of 82 (out of 116) new faults across all groups. In `TerpPresent` we uncovered 119 (out of 126) new faults. As we increased our strength of testing, the number of faults detected usually increased within individual groups. For instance we found 5 new faults in Group 3 of `TerpPaint` at t=2 and 8 faults at t=3. `TerpSpreadSheet` and `TerpWord` show similar results for a total of 66 (out of 114) and 40 (out of 71) new faults respectively. In `TerpWord` we found new faults only in a small number of groups (1-3) but the trends for those groups hold. Although an increase in covering array strength usually increased the number of faults we did see some exceptions. For instance, in `TerpSpreadsheet`, Group 6, we found 10 faults when t=3; however, for t=4, we found only 9 of the 10 previously found faults. We attribute this to chance; a specific longer sequence needed to detect this fault happened to appear in the lower strength test suite. Figure 4 shows the cumulative fault coverage across all groups by covering array strength for the `TerpOffice` applications. The last column for each subject is the t-10-*covering array* fault detection, labeled CA. The other columns represent the other adequacy criteria and are discussed later.

In the `SourceForge` applications we found new faults as well, but not as many as were found in the `TerpOffice` applications. This is expected since they are *real faults*, and only detected by crashes, rather than seeded ones detected by oracles. Table 3 shows the results for each of these applications. In `CrosswordSage` and `JMSN` no new faults were found, but in `FreeMind` we uncovered 4 new faults and in `GanttProject` we detected 7. Once again, the strength of the test suite seems to correlate with the ability to find new faults. For instance in `GanttProject` we found no new faults in Group 1 when t=2 but found 3 new faults when t=3. Figure 5 shows the cumulative fault coverage for all of the `SourceForge` applications.



(a) `TerpSpreadSheet` Group 6 and Group 1

(b) `TerpPresent` Group 1 and Group 8

(c) `Freemind` Group 3 and Group 6

(d) `GanttProject` Group 5 and Group 2

Fig. 6. Adequacy of Covering Array Test Suites

5.2 Adequacy of Test Suites

The next part of our characterization analyzes the coverage of our test suites based on the context based adequacy criteria developed in Section 3. We have seen that more faults are found in higher strength covering arrays, but in some cases we find that increasing strength does not improve fault detection at all or only does so slightly. To obtain more insight into this we examine the adequacy of the covering array test suites.

Figure 6 shows data for two `TerpOffice` applications and two `SourceForge` applications. In each graph we have selected two groups to show. We see similar trends in the other applications/groups. We measure the adequacy up to t+1 where t is the highest strength test suite in a group. For instance `TerpSpreadSheet` Group 6 (Figure 6(a)) was tested up to t=5, therefore we show coverage up through t=6 . In Group 1 (right portion of the same graph) we tested up to t=3 so we show coverage up to t=4. On the `x-axis` we list the t^+-$cover$ ($+$), t-$cover$ (cov), t^*-$cover$ ($*$), followed by the *t-k-covering array* (CA) adequacy, for each t, while on the `y-axis` we show the percent adequacy. Each of the delineated regions on the `x-axis` represents a single t-10-*covering array* test suite, labeled as Gx:TCAt=y where x stands for the group number and y stands for the strength of testing; G2:TCAt=2 is a t=2 array for Group 2.

We see that even in the case where we had many test cases fail and our t-k-*covering array* adequacy is quite low, we have a high percentage of t^+-*cover* and t^*-*cover* coverage. For instance if we examine the adequacy for t=4 for `TerpPresent` (Figure 6(b)) In the 3-10-*covering arrays*, we see that in both groups (second and last regions of this graph) our *4-10-covering array* adequacy is very low (less than 20%), but we have 100% t^+-*cover* adequacy in the same arrays. We see lower t-$cover$ adequacy and a mid-range for t^*-*cover* adequacy. This is not as obvious as the other results, as it is a stronger criterion then t-$cover$, but since t^*-*cover* adequacy contains t^+-*cover* adequacy within it, and this has reached 100% the overall adequacy is not as low as the t-$cover$. If we examine the 2-10-*covering arrays* for both groups (first and third regions of the same graph) we have less than 5% 4-10-*covering array* adequacy but more than 60% t^+-*cover* adequacy. Once again the t^*-*cover* coverage is between the t^+-*cover* and t-*cover* adequacy.

In Figure 6(c), `FreeMind` and Figure 6(d) `GanttProject`, we see that the groups had slightly different overall adequacies, but the general trends are the same as that of the `TerpOffice` applications. Figure 6(d) shows results for `GanttProject`. In this subject, Group 2 has an overall low adequacy, only reaching 50% 2^+ coverage and less than 20% 2-10-*covering array* adequacy.

In general we can see that the t^+-*cover* coverage is the highest, followed by t^*-*cover*, followed by t-*cover*. This makes sense since there are combinatorially more opportunities to cover an *event-non-consecutive-t-sequence* than an *event-consecutive-t-sequence* in a fixed length test sequence. It also suggests that the weakest test criteria (the easiest to cover) is t^+-*cover*, while the strongest is the t-10-*covering array*. This provides us with a characterization of the adequacy obtained within our original test suites, but it does not provide us information about the relative strength of fault detection for each criteria. We examine that next.

6 Fault Detection Results

We can see from the characterization of our test suites that $(t+1)^+$ and $t+1$-$cover$ is often higher than the associated t-10-*covering* array adequacy which means they are easier to satisfy and may be potentially useful for testing. But we do not yet know if these test suites are effective at finding faults. We now analyze the fault detection by each of the adequacy criteria from the test suites derived in Step 6, and then discuss some of the specific faults detected by our test cases.

6.1 Fault Detection and Adequacy Criteria

Figures 7 and 8 show the average fault detection by coverage criteria for some of our subjects. For example, the upper-left graph in Figure 7 shows the results for `TerpWord`, Group 3. The y-axis shows the average number of faults detected across all five samples; the x-axis shows the criteria for each strength. The graph clearly shows that t=3 coverage outperforms its t=2 counterparts. It also confirms that t^+-*cover* adequate test suites have the lowest fault detection while the t-10-*covering array* has the highest fault detection. t^*-*cover* is the next strongest criteria while, t-$cover$ falls in-between the others.

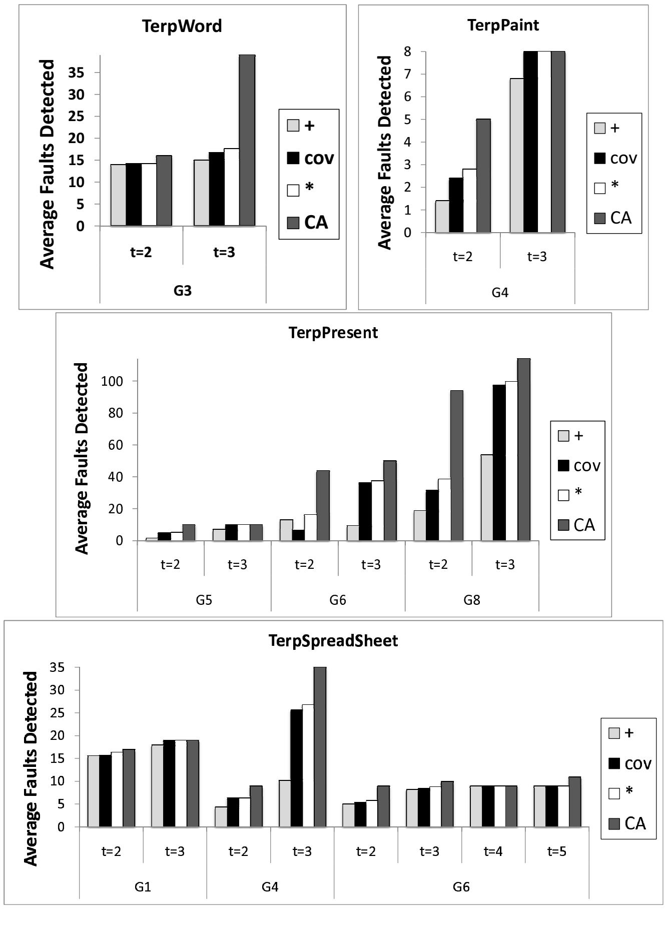


Fig. 7. Fault Detection by Coverage Criteria (TerpOffice)

The other graphs (including FreeMind in Figure 8) show more or less similar trends. This is consistent with our expectations for the adequacy criteria. In some cases we see that using a medium strength criteria, t-$cover$ or t^*-*cover* provides almost the same fault detection as using a t-10-*covering array* (TerpPresent Group 5, TerpSpreadsheet Group 1, TerpPaint Group 4 (when t=3) and Freemind Group 3 and 6). We examine the tradeoff of size below.

We see some small anomalies in the graphs where the t^+-*cover* adequate test suites find more faults than the $(t+1)^+$-*cover* adequate suites. This includes TerpPresent Group 6 and and FreeMind, Group 3. We also see an inversion for t-10-*covering arrays* between t=3 and t=4 for TerpSpreadSheet, Group 6. These may be due to the greedy method we used to select the test suites. Upon examining some of the $t+1$ test suites we have found a single test case that is shared between several of the test suites for the lower coverage criteria that finds a large number of faults.

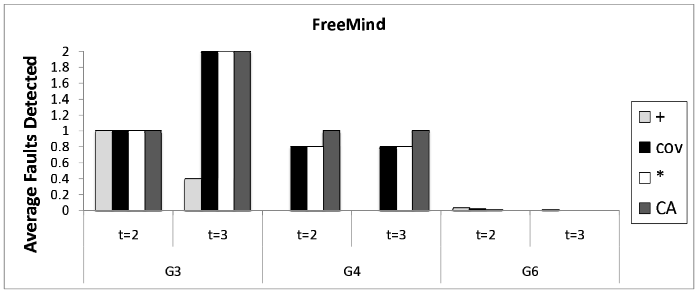


Fig. 8. Fault Detection by Coverage Criteria (FreeMind)

Figures 4 and 5, presented earlier, provide cumulative fault coverage across groups for each of the adequacy criteria. The data is consistent with the individual group coverage.

The last part of our analysis examines the test suite size of the various criteria. Tables 4 and 5 show data for our subjects. In these tables we show each subject, each group and the average fault detection and test suite size for each coverage criteria at each strength. For instance in TerpPaint we can see that the t^+-*cover* test suite for t=2 in Group 1 contains only 43.8 test cases on average, while the t-$cover$ contains almost 4 times as many (123.4). While the least expensive criterion to satisfy is t^+-*cover*, we also lose effectiveness. In this same subject/group, we see that the t^+-*cover* only detected 19 faults on average while the t-$cover$ detected 33. One interesting observation is that the t^*-$cover$ suites find more faults than the t-$cover$ in some cases. For instance in TerpPresent, Group 8, we see this trend for both for t=2 and t=3. We find on average 38.6 vs. 31.6 faults at t=2 when comparing t^*-*cover* and t-$cover$ and 99.6 vs. 97.6 for t=3. This suggests that t^+-*cover* and t^*-$cover$ adequate test suites find different sets of faults.

In the SourceForge applications (Table 5) the fault detection differences are not as dramatic. In fact, in GanttProject, Group 1, when t=2 we find the same 3 faults using t^+-*cover* for an average of 169 test cases, as we do for the 2-10-*covering array* which has almost 6,000 test cases. Only a couple of groups show weaker fault detection

Group	Strength	t^+-*cover*	t-*cover*	t^*-*cover*	t-10-CA
TerpPaint					
Group 1	t=2	19.0/43.8	33.0/123.4	38.0/161.8	69/1055
Group 2	t=2	3.4/73.6	4.0/208.6	4.0/273.4	4/1783
Group 3	t=2	1.4/8.8	2.4/22.6	2.8/28.8	5/171
	t=3	6.8/40.2	8.0/255.8	8.0/291.8	8/2870
Group 5	t=4	5.0/22.0	5.0/81.4	5.0/84.2	5/3428
TerpPresent					
Group 1	t=2	0.0/13.0	0.2/33.6	0.2/45.4	1/280
	t=3	0.0/74.4	1.4/494.4	1.4/562.0	5/5964
Group 2	t=2	0.0/24.8	0.0/66.8	0.0/89.8	1/578
	t=3	0.2/207.6	4.0/1428.6	4.0/1621.2	5/19214
Group 5	t=2	1.6/14.4	5.0/36.4	5.2/45.8	10/280
	t=3	7.0/83.4	10.0/555.2	10.0/623.0	10/5964
Group 6	t=2	13.0/8.0	6.4/21.0	16.4/28.0	44/171
	t=3	9.4/38.4	36.2/239.0	37.4/274.0	50/2870
Group 8	t=2	18.8/6.4	31.6/17.0	38.6/22.6	94/144
	t=3	53.8/28.8	97.6/181.8	99.6/209.2	114/2133
TerpSpreadSheet					
Group 1	t=2	15.6/12.6	15.8/33.8	16.4/44.2	17/280
	t=3	18.0/75.0	19.0/498.8	19.0/568.8	19/5964
Group 2	t=2	1.0/9.6	1.0/25.4	1.0/34.2	1/206
	t=3	1.0/48.8	1.0/313.6	1.0/358.2	1/3749
Group 3	t=2	2.0/2.0	2.0/2.6	2.0/4.4	2/25
	t=3	2.0/3.0	2.0/11.2	2.0/13.8	2/131
	t=4	2.0/7.4	2.0/49.4	2.0/56.6	2/639
	t=5	2.0/23.8	2.0/222.8	2.0/244.0	2/2777
Group 4	t=2	4.4/14.0	6.4/30.8	6.4/39.2	9/206
	t=3	10.2/64.4	25.6/352.6	26.8/392.8	35/3749
Group 5	t=2	1.0/5.0	1.0/11.8	1.0/16.0	1/92
	t=3	1.0/16.0	1.0/94.4	1.0/109.2	1/1081
	t=4	1.0/83.0	1.0/814.6	1.0/891.2	1/11454
Group 6	t=2	5.0/2.0	5.4/4.8	5.8/6.0	9/37
	t=3	8.2/4.2	8.4/22.8	8.8/26.4	10/263
	t=4	9.0/16.6	9.0/127.6	9.0/142.6	9/1613
	t=5	9.0/75.4	9.0/727.8	9.0/794.8	11/8977
TerpWord					
Group 1	t=3	0.4/81.0	1.0/535.0	1.0/607.0	1/5964
Group 2	t=2	1.0/64.4	1.0/179.4	1.0/233.6	1/1432
Group 3	t=2	14.0/17.0	14.2/44.2	14.2/58.6	16/363
	t=3	15.6/112.0	16.8/750.2	17.6/851.0	39/10942

TABLE 4
TerpOffice: Average Fault Detection / Test Suite Size

Group	Strength	t^+-*cover*	t-*cover*	t^*-*cover*	t-10-CA
FreeMind					
Group 3	t=2	1.0/26.0	1.0/54.2	1.0/60.2	1/363
	t=3	0.4/166.2	2.0/833.4	2.0/879.8	2/10942
Group 4	t=2	0.0/6.0	0.8/14.8	0.8/19.0	1/144
	t=3	0.0/17.0	0.8/114.0	0.8/125.4	1/2133
Group 6	t=2	1.0/28.6	1.0/65.8	1.0/76.0	1/412
	t=3	1.0/187.0	3.0/1020.6	3.0/1077.6	3/11168
GanttProject					
Group 1	t=3	3.0/168.6	3.0/645.8	3.0/692.6	3/5964
Group 2	t=2	1.0/22.0	0.8/34.0	1.0/40.4	1/280
	t=3	1.0/76.2	1.0/200.4	1.0/224.4	1/5964
Group 3	t=3	1.0/40.8	1.4/205.8	1.4/214.6	2/4851
Group 4	t=3	2.0/31.8	1.6/208.0	2.0/230.2	2/3749

TABLE 5
SourceForge: Average Fault Detection / Test Suite Size

using t^{+}-*cover*. We see only a minor improvement using 2-10-*covering array* adequacy over t-$cover$ leading us to the conclusion that for these subjects the weaker test criteria may be a better choice since they are almost as effective for a fraction of the cost.

6.2 Analysis of Faults Detected

To better understand the role of context on fault detection and adequacy, we analyzed the faults that were detected only by the covering array test cases. This section provides details of our analysis and findings.

FreeMind: As mentioned in Section 1, in `FreeMind 0.8.0`, initially launched with an existing mindmap, containing at least one node, loaded from a file, events e_1: *SetNodeToCloud*, e_2: *NewChildNode*, and e_3: *NewParentNode*, executed in a sequence (either as $<e_1, e_2, e_3>$ or $<e_2, e_1, e_3>$) result in an *ArrayIndexOutOfBoundsException* (in line 124 of `ConvexHull.java`). Detection of this fault requires the execution of at least these three events in a specific order, without certain interleaving events. The exception is thrown when the code attempts a Graham scan to calculate the convex hull (in method `doGraham()` in `ConvexHull.calculateHull`) when painting a currently selected cloud node's view during the creation of its new parent. Hence, e_1 is needed to change the existing node's view to cloud. Event e_2 is needed to ensure that the currently selected node is a child node; event e_3 is then performed on this selected node, which attempts to repaint the map and update the MapView. During the repainting, it invokes the `paintChildren()` method, which in turn invokes the `paintClouds()` method. In this method, during painting, an incorrect value of a variable `M` used to index an array throws the exception. Hence, the exception is thrown only when creating a parent for a currently selected child node in a mindmap that also contains a cloud node.

If another event (*e.g.*, *Undo*, *CloseMap*, *CreateNewMap*, etc.) occurs somewhere in the exception-causing sequence, the new event may cause any number of changes to the GUI's state, preventing the fault from being detected – the child node may no longer be selected (if the root node is selected), it may no longer exist (if the child node is deleted or the previous operation is undone), the map may be closed (on a *Close* event), and so on.

This fault is detected by the t=2 and t=3 test suites for `FreeMind` Group 3. The fact that the t=3 suite detected it is not surprising. However, the t=2 also detected it is interesting. Because all our test cases are of length 10, this suite also covers some 3-way interactions, as evident by the G3:TCA$_{t=2}$ section in Figure 6(c), one of them leading to the exception.

GanttProject: In `GanttProject 2.0.1`, events e_1: *NewResource* (event *ImportResource* also works here), e_2: *NewTask*, and e_3: *NewTask*, executed in a sequence $<e_1, e_2, e_3>$ result in a *NullPointerException* (Line 460 of `GanttTree.java`) for certain large projects. The exception is thrown only when a second *NewTask* event is fired before `GanttProject` has completely handled the first *NewTask* event; this only happens for large projects that impose a significant performance overhead on the underlying Java Virtual Machine. During the execution of the exception-causing sequence, e_1 loads and sets user focus on a "resource tree." The event handler for *NewTask* creates a `Task` object and then updates a "task tree." In doing so, it invokes the `setEditingTask()` method to update the current editing task in the task tree, where it creates a new `TreePath` for the currently selected task. The user focus in the task tree is on the new task. A second *NewTask* event performed before the previous updates (being handled by a different thread) are complete fails to switch focus to the newly formed tree node; invocation of the `getSelectedTaskNode()` method from `setEditingTask()` returns null, throwing a *NullPointerException* on subsequent accesses to this object's fields.

This example highlights the need for incorporating timing information in our GUI test cases – this is a subject for future work. It also shows how combining certain events together is absolutely needed to detect certain faults. If these events had not been executed consecutively, the fault would have been missed.

TerpPaint: In `TerpPaint`, one set of faults (86, 87, 88) was detected by several test cases in our covering array sample. These faults are all found in the handler for the event *SelectEraserTool* which corresponds to the method `eraserActionPerformed()`. The faults incorrectly change an "==" to an "!=" in three different conditions in this handler to check *curZoom*, a property that will decide what type of cursor is to be used for the eraser tool. If *curZoom == zoom2*, for example, the eraser cursor's size will be set to one size, but when *curZoom == zoom1*, it will be set to a different size. When the condition results are incorrectly returned, and the eraser tool is used, different results will occur. One of the test cases from the covering array sample in Group 1 which detects this fault contains the following *abstract* event sequence: <*TextTool*, *LineTool*, *FillWithColor*, *SelectDoubleZoom*, *EraserTool*, *FillWithColor*, *LineTool*, *MoveMouseInCanvas*, *ShowToolBar*, *EllipseTool*>.

Upon examining the code, we realized that detection requires more than three events. First the test case must reach the faulty statements in the code. Two events are needed for this: 1) *SelectAZoomTool* (There are four possibilities which correspond to *zoom1*, *zoom2*, *zoom3*, *zoom4* for *curZoom* in the code.); 2) *SelectEraserTool*. The order of these two events can not be changed. Simply reaching this code is not enough for detection however. The faulty behavior needs to show itself in the GUI for detection by our GUI-based test oracle. In this case, an image is needed where the *EraserTool* can be applied, and the wrong eraser will wipe out a different part of the image. By checking the resulting image on the canvas, one can detect the fault. Here, at least two more events are needed, that is, one for setting up an image and the other for using the eraser tool (*FillWithColor* and *MoveMouseInCanvas* in our test case). Therefore, the shortest sequence that can detect this fault would be a length four sequence. In our experiments, the GUI is started with no image (a white canvas). In the detecting test case the *FillWithColor* event fills the empty canvas with the default color, which is black. After performing the *EraserTool* event, it moves the mouse on the canvas with the eraser tool, then, a black area will be removed (turns white). As the type (therefore the size) of eraser is incorrectly set by the fault, the resulting image is different from the expected one and the test case detects the fault.

7 Conclusions and Future Work

This paper presented a new family of test adequacy criteria for GUI testing. The new criteria are unique in that they allow for "context" in GUI test cases in terms of event combination strength, sequence length, and all possible positions for each event. A case study on eight applications showed that increasing event combination strength and controlling starting and ending positions of events helps to detect a large number of previously undetected faults. We abstracted our event space into a new model, a system interaction event set, that does not have strict ordering constraints, so that we could leverage ideas from CIT for sampling. We then relaxed the CIT criteria to define a family of criteria with increasing coverage and cost. We believe that an important part of our future work should examine more closely the exact cost tradeoff for the various type of criteria.

This work has raised many interesting questions that point to future research. In fact, we consider this as a promising starting point. We are currently examining new techniques to generate test cases that can systematically improve coverage, as specified by our new criteria; in this paper we used subsets of larger test suites rather than directly generated the test cases. The covering array based test case generation approach provided a good starting point, but unexecutable parts of test cases suggest that our approach needs to be augmented. In some groups, the events seem to be extremely interdependent despite our best modeling attempts; allowable sequences require precise execution orders. For such groups, we will manually construct state-machines, instead of automatically reverse engineered event-interaction graphs and we are exploring a dynamic adaptive approach through the use of evolutionary algorithms [44] to discover and repair our constraints during iterative generation and execution.

All of our test cases in the case study were fixed to length 10. In the future, we will vary this length and study the impact of test-case length on faults and coverage. Finally, we currently manually partition the GUI events into groups. Future work will study the automatic partitioning of events, *e.g.*, based on how events change the GUI's state [19].

Acknowledgments

We would like to thank the anonymous reviewers for all of their feedback, insights and comments on this paper. This work was partially supported by the US National Science Foundation under grants CCF-0747009, CCF-0447864, CNS-0855139 and CNS-0855055, the Air Force Office of Scientific Research through award FA9550-09-1-0129, the Office of Naval Research grant N00014-05-1-0421 and by the Defense Advanced Research Projects Agency through award HR0011-09-0031. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the position or policy of NSF, ONR, AFOSR or DARPA.

References

[1] A. P. Mathur, *Foundations of Software Testing: Fundamental Algorithms and Techniques*. Pearson Education., 2008.

[2] Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing," *ACM Transactions on Software Engineering and Methodology*, pp. 1–35, 2008.

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

[4] A. M. Memon and Q. Xie, "Using transient/persistent errors to develop automated test oracles for event-driven software," in *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 186–195.

[5] A. M. Memon, "Developing testing techniques for event-driven pervasive computing applications," in *Proceedings of The OOPSLA 2004 workshop on Building Software for Pervasive Computing (BSPC 2004)*, Oct. 2004.

[6] G. J. Tretmans and H. Brinksma, "TorX: Automated model-based testing," in *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, A. Hartman and K. Dussa-Ziegler, Eds., December 2003, pp. 31–43.

[7] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," in *Formal Methods and Testing, LNCS 4949*, 2008, pp. 39–76.

[8] H. Ural and B. Yang, "A test sequence selection method for protocol testing," *IEEE Transactions on Communications*, vol. 39, no. 4, pp. 514–523, 1991.

[9] A. Marchetto and P. Tonella, "Search-based testing of Ajax web applications," in *1st International Symposium on Search Based Software Engineering*, May 2009, pp. 3–12.

[10] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software Testing, Verification and Reliability*, vol. 13, no. 1, pp. 25–53, 2003.

[11] P. Brooks, B. Robinson, and A. M. Memon, "An initial characterization of industrial graphical user interface systems," in *ICST 2009: Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2009.

[12] "FreeMind - free mind-mapping software," 2009, http://freemind.sourceforge.net.

[13] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," in *European Software Engineering Conference / Foundations of Software Engineering (ESEC/FSE)*, 2001, pp. 256–267.

[14] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[15] X. Yuan, M. Cohen, and A. M. Memon, "Covering array sampling of input event sequences for automated GUI testing," in *International Conference on Automated Software Engineering (ASE)*, 2007, pp. 405–408.

[16] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG system: an approach to testing based on combinatorial design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, pp. 437–444, 1997.

[17] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge, "Constructing test suites for interaction testing," in *International Conference on Software Engineering (ICSE)*, May 2003, pp. 38–48.

[18] J. Strecker and A. M. Memon, "Relationships between test suites, faults, and fault detection in GUI testing," in *First international conference on Software Testing, Verification, and Validation (ICST)*, 2008, pp. 12–21.

[19] X. Yuan and A. M. Memon, "Using GUI run-time state as feedback to generate test cases," in *International Conference on Software Engineering (ICSE)*, 2007, pp. 396–405.

[20] F. J. Daniels and K. C. Tai, "Measuring the effectiveness of method test sequences derived from sequencing constraints," in *Technology of Object-Oriented Languages and Systems (TOOLS)*, 1999, pp. 74–83.

[21] U. Farooq, C. P. Lam, and H. Li, "Towards automated test sequence generation," in *Australian Software Engineering Conference*, 2008, pp. 441–450.

[22] K. Inkumsah and T. Xie, “Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution,” in *Automated Software Engineering (ASE)*, 2008, pp. 297–306.

[23] L. Gallagher and J. Offutt, “Test sequence generation for integration testing of component software,” *The Computer Journal*, pp. 1–16, 2007.

[24] A. Gargantini and E. Riccobene, “ASM-based testing: Coverage criteria and automatic test sequence generation,” *Joual of Universal Computer Science.*, vol. 7, no. 11, pp. 1050–1067, 2001.

[25] R. K. Shehady and D. P. Siewiorek, “A method to automate user interface testing using variable finite state machines,” in *International Symposium on Fault -Tolerant Computing (FTCS)*, 1997, pp. 80–88.

[26] A. M. Memon, M. E. Pollack, and M. L. Soffa, “Hierarchical GUI test case generation using automated planning,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 144–155, 2001.

[27] L. White and H. Almezen, “Generating test cases for GUI responsibilities using complete interaction sequences,” in *International Symposium on Soft ware Reliability Engineering (ISSRE)*, 2000, pp. 110–121.

[28] S. McMaster and A. M. Memon, “Call-stack coverage for GUI test-suite reduction,” *IEEE Transactions on Software Engineering*, 2008.

[29] Q. Xie and A. M. Memon, “Rapid “crash testing” for continuously evolving GUI-based software applications,” in *International Conference on Software Maintenance (ICSM)*, 2005, pp. 473–482.

[30] F. Belli, “Finite-state testing and analysis of graphical user interfaces,” in *International Symposium on Software Reliability Engineering (ISSRE)*, 2001, pp. 34–43.

[31] L. J. White, “Regression testing of GUI event interactions,” in *International Conference on Software Maintenance (ICSM)*, 1996, pp. 350–358.

[32] “JUnit, Testing Resources for Extreme Programming,” http://junit.org/news/extension/gui/index.htm.

[33] “Mercury Interactive WinRunner,” 2003, http://www.mercuryinteractive.com/products/winrunner.

[34] R. Brownlie, J. Prowse, and M. S. Phadke, “Robust testing of AT&T PMX/StarMAIL using OATS,” *AT& T Technical Journal*, vol. 71, no. 3, pp. 41–47, 1992.

[35] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, “Applying design of experiments to software testing,” in *International Conference on Software Engineering, (ICSE)*, 1997, pp. 205–215.

[36] D. Kuhn, D. R. Wallace, and A. M. Gallo, “Software fault interactions and implications for software testing,” *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 418–421, 2004.

[37] C. Yilmaz, M. B. Cohen, and A. Porter, “Covering arrays for efficient fault characterization in complex configuration spaces,” *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 20–34, 2006.

[38] X. Qu, M. B. Cohen, and G. Rothermel, “Configuration-aware regression testing: An empirical study of sampling and prioritization,” in *International Symposium on Software Testing and Analysis (ISSTA)*, July 2008, pp. 75–85.

[39] M. B. Cohen, M. B. Dwyer, and J.Shi, “Coverage and adequacy in software product line testing,” in *Workshop on the Role of Architecture for Testing and Analysis (ROSATEA)*, July 2006, pp. 53–63.

[40] D. Chays, S. Dan, Y. Deng, F. I. Vokolos, P. G. Frankl, and E. J. Weyuker, “AGENDA: A test case generator for relational database applications,” Polytechnic University, Tech. Rep., 2002.

[41] R. C. Bryce, A. Rajan, and M. P. E. Heimdahl, “Interaction testing in model-based development: Effect on model-coverage,” in *Asia Pacific Software Engineering Conference (ASPEC)*, 2006, pp. 259–268.

[42] Q. Xie and A. M. Memon, “Model-based testing of community-driven open-source GUI applications,” in *International Conference on Software Maintenance (ICSM)*, 2006, pp. 145–154.

[43] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[44] X. Yuan, M. Cohen, and A. M. Memon, “Towards dynamic adaptive automated test generation for graphical user interfaces,” in *First International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS)*, 2009, pp. 1–4.

Xun Yuan is a Software Engineer in Test (SET) at Google Kirkland where she is in charge of ensuring the quality of a web-based software product called Website Optimizer. She completed her PhD from the Department of Computer Science at the University of Maryland in 2008 and MS in Computer Science from the Institute of Software Chinese Academy of Sciences in 2001. Her research interests include software testing, quality assurance, web application design, and model-based design. In addition to her interests in Computer Science, she also likes mathematics and literature.

Myra B. Cohen is an Assistant Professor in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln where she is a member of the Laboratory for Empirically based Software Quality Research and Development (ESQuaReD). She received the PhD degree in computer science from the University of Auckland, New Zealand and the MS degree in computer science from the University of Vermont. She received the BS degree from the School of Agriculture and Life Sciences, Cornell University. She is a recipient of a National Science Foundation Faculty Early CAREER Development Award and an Air Force Office of Scientific Research Young Investigator Program Award. Her research interests include testing of configurable software systems and software product lines, combinatorial interaction testing, and search based software engineering. She is a member of the IEEE and ACM.

Atif M Memon is an Associate Professor at the Department of Computer Science, University of Maryland. His research interests include program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He is the inventor of the GUITAR system (http://guitar.sourceforge.net/) for automated model-based GUI testing. He is the founder of the International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS). He serves on various editorial boards, including that of the Journal of Software Testing, Verification, and Reliability. He has served on numerous National Science Foundation panels and program committees, including ICSE, FSE, ICST, WWW, ASE, ICSM, and WCRE. He is currently serving on a National Academy of Sciences panel as an expert in the area of Computer Science and Information Technology, for the Pakistan-U.S. Science and Technology Cooperative Program, sponsored by United States Agency for International Development (USAID). In addition to his research and academic interests, he handcrafts fine wood furniture.