Coverage Criteria for GUI Testing

Atif M. Memon*
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
atif@cs.pitt.edu

Mary Lou Soffa†
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
soffa@cs.pitt.edu

Martha E. Pollack‡
Dept. of Electrical Engineering
and Computer Science,
University of Michigan
Ann Arbor, MI 48109
pollackm@eecs.umich.edu

ABSTRACT

The widespread recognition of the usefulness of graphical user interfaces (GUIs) has established their importance as critical components of today's software. GUIs have characteristics different from traditional software, and conventional testing techniques do not directly apply to GUIs. This paper's focus is on **coverage criteria** for GUIs, important rules that provide an objective measure of test quality. We present new coverage criteria to help determine whether a GUI has been adequately tested. These coverage criteria use events and event sequences to specify a measure of test adequacy. Since the total number of permutations of event sequences in any non-trivial GUI is extremely large, the GUI's hierarchical structure is exploited to identify the important event sequences to be tested. A GUI is decomposed into *GUI components*, each of which is used as a basic unit of testing. A representation of a GUI component, called an *event-flow graph*, identifies the interaction of events within a component and, *intra-component criteria* are used to evaluate the adequacy of tests on these events. The hierarchical relationship among components is represented by an *integration tree*, and *inter-component coverage criteria* are used to evaluate the adequacy of test sequences that cross components. Algorithms are given to construct event-flow graphs and an integration tree for a given GUI, and to evaluate the coverage of a given test suite with respect to the new coverage criteria. A case study illustrates the usefulness of the coverage report to guide further testing and an important correlation between event-based coverage of a GUI and statement coverage of it's software's underlying code.

*Partially supported by the Andrew Mellon Pre-doctoral Fellowship. Effective Aug 1, 2001: Department of Computer Science, University of Maryland. atif@cs.umd.edu

†Partially supported by a grant from the National Science Foundation (CCR 9808590 and EIA 9806525) to the University of Pittsburgh.

‡Partially supported by the Air Force Office of Scientific Research (F49620-01-1-0066) and Defense Advanced Research Projects Agency (F30602-00-2-0621)

Keywords

GUI testing, GUI test coverage, Event-based coverage, Event-flow graph, Integration tree, Component testing.

1. INTRODUCTION

The importance of graphical user interfaces (GUIs) as critical components of today's software is increasing with the recognition of their usefulness. The widespread use of GUIs has led to the construction of more and more complex GUIs. Although the use of GUIs continues to grow, GUI testing has, until recently, remained a neglected research area. Because GUIs have characteristics different from conventional software, techniques developed to test conventional software cannot be directly applied to GUI testing. Recent advances in GUI testing have focused on the development of test case generators [8, 11, 12, 14, 18, 6] and test oracles [9] for GUIs. However, development of *coverage criteria* for GUIs has not been addressed.

Coverage criteria are sets of rules to help determine whether a test suite has adequately tested a program and to guide the testing process. The most well-known coverage criteria are structural, and include statement coverage, branch coverage, and path coverage, which require that every statement, branch and path in the program's code be executed by the test suite respectively. However such criteria do not address the adequacy of GUI test cases for a number of reasons. First, GUIs are typically developed using instances of precompiled elements stored in a library. The source code of these elements may not always be available for coverage evaluation. Second, the input to a GUI consists of a sequence of events. The number of possible permutations of the events may lead to a large number of GUI states and for adequate testing, a GUI event may need to be tested in a large number of these states. Moreover, the event sequences that must be tested on the GUI are conceptually at a much higher level of abstraction than the code and hence cannot be obtained from the code. For the same reason, the code cannot be used to determine whether an adequate number of these sequences has been tested on the GUI.

The above challenges suggest the need to develop coverage criteria based on **events** in a GUI. The development of such coverage criteria has certain requirements. First, since there are a large number of possible permutations of GUI events, the GUI must be decomposed into manageable parts. GUIs, by their very nature, are hierarchical and this hierarchy may

be exploited to identify groups of GUI events that can be tested in isolation. Hence, each group forms a *unit of testing*. Such a decomposition allows coverage criteria to be developed for events within a unit. Intuitively, a unit of testing has a well-defined interface to the other parts of the software. It may be invoked by other units when needed and then terminated. For example, when performing code-based testing, a unit of testing may be a basic block, procedure, an object, or a class, consisting of statements, branches, etc. *Interactions* among units must also be identified and coverage developed to determine the adequacy of tested interactions. Second, it should be possible to satisfy the coverage criterion by a finite-sized test suite. The *finite applicability* [20] requirement holds if a coverage criterion can always be satisfied by a finite-sized test suite. Finally, the test designer should recognize whether a coverage criterion can be fully satisfied [16, 17]. For example, it may not always be possible to satisfy path coverage because of the presence of *infeasible paths*, which are not executable because of the context of some instructions. No test case can execute along an infeasible path, perhaps resulting in loss of coverage. Detecting infeasible paths in general is a NP complete problem. Infeasibility can also occur in GUIs. Similar to infeasible paths in code, static analysis of the GUI may not reveal infeasible sequences of events. For example, by performing static analysis of the menu structure of MS Wordpad, one may construct a test case with `Paste` as the first event. However, experience of using the software shows that such a test case will not execute since `Paste` is highlighted only after a `Cut` or `Copy`.[1]

In this paper, we define a new class of coverage criteria called *event-based coverage criteria* to determine the adequacy of tested event sequences, focusing on GUIs. The key idea is to define the coverage of a test suite in terms of GUI events and their interactions. Since the total number of permutations of event sequences in any non-trivial GUI is extremely large, the GUI's hierarchical structure is exploited to identify the important event sequences to be tested. The GUI is decomposed into *GUI components*,[2] each of which is a unit of testing. Events within a component do not interleave with events in other components without explicit invocation or termination events. Because of this well-defined behavior, a component may be tested in isolation. Two kinds of coverage criteria are developed from the decomposition – *intra-component coverage criteria* for events within a component and *inter-component coverage criteria* for events among components. Intra-component criteria include *event*, *event-interaction*, and *length-n event-sequence* coverage. Inter-component criteria include *invocation*, *invocation-termination* and *length-n event-sequence* coverage. A GUI component is represented by a new structure called an *event-flow graph* that identifies events within a component. The interactions among GUI components are captured by a representation called the *integration tree*. We present algorithms to automatically construct event-flow graphs and the integration tree for a given GUI and to evaluate intra- and inter-component coverage for a given test suite. We present a case study to demonstrate (1) the correlation between event-based coverage of our version of WordPad's GUI and the statement coverage of its underlying code for a test suite, and (2) the usefulness of the coverage report to guide further testing.

The important contributions of the coverage method presented in this paper include:

1. A definition of a GUI **component**, a useful concept for structured GUI testing, and the decomposition of a GUI into a hierarchy of interacting components.
2. a representation of a GUI component called an **event-flow graph** that captures the flow of events within a component and a representation called the **integration tree** to identify interactions among components.
3. a class of coverage criteria for intra-component and inter-component GUI testing and a technique to compute the coverage of a given test suite.
4. a case study demonstrating the usefulness of event-based coverage and a correlation between coverage in terms of events and code.

In the next section we present a classification of GUI events and use the classification to identify GUI components. In Section 3 we present coverage criteria for event interactions within a component and among components. Section 4 presents algorithms to construct event-flow graphs and an integration tree for a given GUI and then evaluate intra- and inter-component coverage of the GUI for a given test suite. In Section 5, we present details of a case study conducted on our version of the WordPad software. Section 6 presents related work, and in Section 7 we conclude with a discussion of ongoing and future work.

2. STRUCTURE OF A GUI

In this section, we first present the class of GUIs that our coverage criteria target. We then describe the structure of GUIs in terms of a hierarchy of components.

2.1 What is a GUI?

A GUI is composed of objects (buttons, menus, trash-can, recycling-bin) using metaphors familiar in real life. The software user interacts with the objects by performing events that manipulate the GUI objects as one would real objects. Events cause deterministic changes to the state of the software that may be reflected by a change in the appearance of one or more GUI objects. Moreover, GUIs, by their very nature, are hierarchical. This hierarchy is reflected in the grouping of events in windows, dialogs, and hierarchical menus. For example, all the "options" in MS Internet Explorer can be set by interacting with events in one window of the software's GUI.

The important characteristics of GUIs include their graphical orientation, event-driven input, hierarchical structure, the objects they contain, and the properties (attributes) of those objects. Formally, we define the class of GUIs of interest as follows:

Definition: A *Graphical User Interface (GUI)* is a hierarchical, graphical front-end to a software that accepts

[1]Note that `Paste` will be available if the ClipBoard is not empty, perhaps because of an external software. External software is ignored in this simplified example.

[2]GUI components should not be confused with *GUI elements* that are used as building blocks during GUI development. We later provide a formal definition of a GUI component.

as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. A GUI contains graphical *objects* and each object has a fixed set of *properties*. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI. □

The above definition specifies a class of GUIs that have a fixed set of events with deterministic outcome that can be performed on objects with discrete valued properties. This definition would need to be extended for other GUI classes such as web-user interfaces that have synchronization/timing constraints among objects, movie players that show a continuous stream of video rather than a sequence of discrete frames, and non-deterministic GUIs in which it is not possible to model the state of the software in its entirety and hence the effect of an event cannot be predicted. This paper develops coverage criteria for the class of GUIs defined above.

2.2 GUI Components and Event Classification

Since today's GUIs are large and contain a large number of events, any scalable representation must decompose a GUI into manageable parts. As mentioned earlier, GUIs are hierarchical, and this hierarchy may be exploited to identify groups of GUI events that can be tested in isolation. One hierarchy of the GUI, and the one used in this paper, is obtained by examining the structure of *modal windows* in the GUI.

Definition: A *modal window* is a GUI window that once invoked, monopolizes the GUI interaction, restricting the focus of the user to a specific range of events within the window, until the window is explicitly terminated. □

The language selection window in MS Word is an example of a modal window. When the user performs the event `Set Language`, a window entitled `Language` opens and the user spends time selecting the language, and finally explicitly terminates the interaction by either performing `OK` or `Cancel`.

Other windows in the GUI are called *modeless windows* that do not restrict the user's focus; they merely expand the set of GUI events available to the user. For example, in the MS Word software, performing the event `Replace` opens a modeless window entitled `Replace`.

At all times during interaction with the GUI, the user interacts with events within a modal dialog. This modal dialog consists of a modal window X and a set of modeless windows that have been invoked, either directly or indirectly by X. The modal dialog remains in place until X is explicitly terminated. Intuitively, the events within the modal dialog form a *GUI component*.

Definition: A *GUI component* C is an ordered pair ($\mathcal{RF}$, $\mathcal{UF}$), where $\mathcal{RF}$ represents a modal window in terms of its events and $\mathcal{UF}$ is a set whose elements represent modeless windows also in terms of their events. Each element of $\mathcal{UF}$ is invoked either by an event in $\mathcal{UF}$ or $\mathcal{RF}$. □

	Event Type					
Component Name	**Menu Open**	**System Interaction**	**Restricted Focus**	**Unrestricted Focus**	**Termination**	**Sum**
Main	7	27	19	2	1	**56**
FileOpen	0	8	0	0	2	**10**
FileSave	0	8	0	0	2	**10**
Print	0	9	1	0	2	**12**
Properties	0	11	0	0	2	**13**
PageSetup	0	8	1	0	2	**11**
FormatFont	0	7	0	0	2	**9**
Sum	**7**	**78**	**21**	**2**	**13**	**121**

Table 1: Types of Events in Some Components of MS WordPad.

An example of a GUI component is the `FileOpen` modal window (and its associated modeless windows) found in most of today's software. The user interacts with events within this component, selects a file and terminates the component by performing the `Open` event (or sometimes the `Cancel` event).

Note that, by definition, events within a component do not interleave with events in other components without the components being explicitly invoked or terminated.

Since components are defined in terms of modal windows, a classification of GUI events is used to identify components. The classification of GUI events is as follows:

Restricted-focus events open *modal windows*. For example, `Set Language`, discussed earlier, is a restricted-focus event.

Unrestricted-focus events open *modeless windows*. For example, `Replace` in MS WordPad is an unrestricted-focus event.

Termination events close modal windows; common examples include `Ok` and `Cancel`.

The GUI contains other types of events that do not open or close windows but make other GUI events available. These events are used to open menus that contain several events.

Menu-open events are used to open menus. They expand the set of GUI events available to the user. Menu-open events do not interact with the underlying software. Note that the only difference between menu-open events and unrestricted-focus events is that the latter open windows that need to be explicitly terminated. The most common example of menu-open events are generated by buttons that open pull-down menus. Common example include `File` and `Edit`.

Finally, the remaining events in the GUI are used to interact with the underlying software.

System-interaction events interact with the underlying software to perform some action; common examples include the `Copy` event used for copying objects to the clipboard.

Table 1 lists some of the components of WordPad. Each row represents a component and each column shows the different

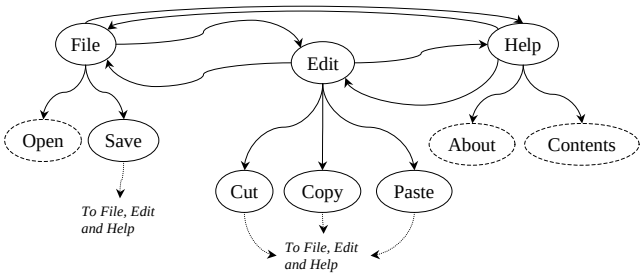

Figure 1: An Event-flow Graph for a Part of MS WordPad.

types of events available within each component. `Main` is the component that is available when WordPad is invoked. Other components' names indicate their functionality. For example, `FileOpen` is the component of WordPad used to open files.

2.3 Event-flow Graphs

A GUI component may be represented as a flow graph. Intuitively, an *event-flow graph* represents all possible interactions among the events in a component.

Definition: An *event-flow graph* for a GUI component C is a 4-tuple <**V**, **E**, **B**, **I**> where:

1. **V** is a set of vertices representing all the events in the component. Each $v \in$**V** represents an event in C.
2. **E** $\subseteq$ **V** $\times$ **V** is a set of directed edges between vertices. We say that event e_i **follows** e_j iff e_j may be performed *immediately* after e_i. An edge $(v_x, v_y) \in$ **E** iff the event represented by v_y **follows** the event represented by v_x.
3. **B** $\subseteq$ **V** is a set of vertices representing those events of C that are available to the user when the component is first invoked.
4. **I** $\subseteq$ **V** is the set of restricted-focus events of the component.

□

An example of an event-flow graph for a part of the `Main`[3] component of MS WordPad is shown in Figure 1. At the top are three vertices (`File`, `Edit`, and `Help`) that represent part of the pull-down menu of MS WordPad. They are menu-open events that are available when the `Main` component is first invoked. Hence they form the set **B**. Once `File` has been performed in WordPad, any of `Edit`, `Help`, `Open`, and `Save` events may be performed. Hence there are edges in the event-flow graph from `File` to each of these events. Note that `Open` is shown with a dashed oval. We use this representation for restricted-focus events, i.e., events that invoke components. Similarly, `About` and `Contents` are also restricted-focus events, i.e., for this component, **I** = {`Open`, `About`, `Contents`}. All other events (`Save`, `Cut`, `Copy`, and `Paste`) are system-interaction events. After any of these

[3]We assume that all GUIs have a `Main` component, i.e., the component that is presented to the user when the GUI is first invoked.

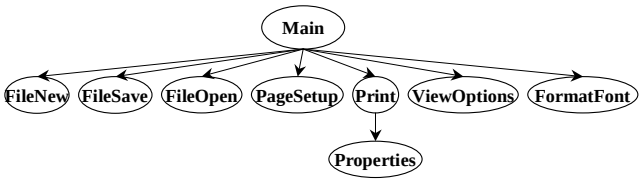

Figure 2: An Integration Tree for a Part of MS WordPad.

is performed in MS WordPad, the user may perform `File`, `Edit`, or `Help`, shown as edges in the event-flow graph.

2.4 Integration Tree

Once all the components of the GUI have been represented as event-flow graphs, the remaining step is to identify their interactions. Testing interactions among components is also an area of research in object-oriented software testing [5] and inter-procedural data-flow testing [4]. The identification of interactions among objects and procedures is aided by structures such as function-decomposition trees and call-graphs [4]. Similarly, we develop a structure to identify interactions among components. We call this structure an *integration tree* because it shows how the GUI components are integrated to form the GUI. Formally, an integration tree is defined as follows:

Definition: An *integration tree* is a 3-tuple $< \mathcal{N}, \mathcal{R}, \mathcal{B} >$, where $\mathcal{N}$ is the set of components in the GUI, $\mathcal{R} \in \mathcal{N}$ is a designated component called the `Main` component. We say that a component C_x **invokes** component C_y if C_x contains a restricted-focus event e_x that invokes C_y. $\mathcal{B}$ is the set of directed edges showing the invokes relation between components, i.e., $(C_x, C_y) \in \mathcal{B}$ iff C_x **invokes** C_y. □

Figure 2 shows an example of an integration tree representing a part of the MS WordPad's GUI. The nodes represent the components of the MS WordPad GUI and the edges represent the invokes relationship between the components. `Main` is the top-level component that is available when WordPad is invoked. Other components' names indicate their functionality. For example, `FileOpen` is the component of WordPad used to open files. The tree in Figure 2 has an edge from `Main` to `FileOpen` showing that `Main` contains an event, namely `Open` (see Figure 1) that invokes `FileOpen`.

3. COVERAGE CRITERIA

Having created representations for GUI components and events within components, we are ready to define the coverage criteria. We will first define coverage criteria for events within a component, i.e., *intra-component coverage criteria* and then for events among components, i.e., *inter-component criteria*.

3.1 Intra-component Coverage

In this section, we define several coverage criteria for events and their interactions within a component. We first formally define an *event sequence*.

Definition: An *event-sequence* is $< e_1, e_2, e_3, ..., e_n >$ where $(e_i, e_{i+1}) \in \mathbf{E}$, $1 \leq i \leq n-1$. □

All the new coverage criteria that we define next are based on event-sequences.

3.1.1 Event Coverage

Intuitively, event coverage requires each event in the component to be performed at least once. Such a requirement is necessary to check whether each event executes as expected.

Definition: A set P of event-sequences satisfies the *event coverage criterion* if and only if for all events $v \in \mathbf{V}$, there is at least one event-sequence $p \in P$ such that event v is in p. □

3.1.2 Event-interaction Coverage

Another important aspect of GUI testing is to check the interactions among all possible pairs of events in the component. However, we want to restrict the checks to pairs of events that may be performed in a sequence.

Definition: The *event-interactions* for an event e is the set $\{e_j | (e, e_j) \in \mathbf{E}\}$. □

In this criterion, we require that after an event e has been performed, all events that can interact with e should be executed at least once. Note that this requirement is equivalent to requiring that each element in $\mathbf{E}$ be covered by at least one test case.

Definition: A set P of event-sequences satisfies the *event-interaction coverage criterion* if and only if for all elements $(e_i, e_j) \in \mathbf{E}$, there is at least one event-sequence $p \in P$ such that p contains (e_i, e_j). □

3.1.3 Length-n Event-sequence Coverage

In certain cases, the behavior of events may change when performed in different contexts. In such cases event coverage and event-interaction coverage on their own are weak requirements for sufficient testing. We now define a criterion that captures the contextual impact. Intuitively, the context for an event e is the sequence of events performed before e. An event may be performed in an infinite number of contexts. For finite applicability, we define a limit on the length of the event-sequence. Hence, we define the *length-n event-sequence criterion.*

Definition: A set P of event-sequences satisfies the *length-n event-sequence coverage criterion* if and only if P contains all event-sequences of length equal to n. □

Note the similarity of this criterion to the *length-n path coverage criterion* defined by Gourlay for conventional software [2], which requires coverage of all subpaths in the program's flow-graph of length less than or equal to n. As the length of the event-sequence increases, the number of possible contexts also increases.

3.2 Subsumption

A coverage criterion $\mathcal{C}_\infty$ **subsumes** criterion $\mathcal{C}_\in$ if every test suite that satisfies $\mathcal{C}_\infty$ also satisfies $\mathcal{C}_\in$ [13]. Since event coverage and event-interaction coverage are special cases of length-n event-sequence coverage, i.e., length 1 event-sequence and length 2 event-sequence coverage respectively, it follows that length-n event-sequence coverage **subsumes** event and event-interaction coverage. Moreover, if a test suite satisfies event-interaction coverage, it must also satisfy event coverage. Hence, event-interaction **subsumes** event coverage. The **subsume** relationship between the coverage criteria is summarized in Figure 3. The nodes represent the criteria whereas the edges represent the subsume relation. Note that the figure also shows inter-component coverage criteria (in reverse color). The relationships among these criteria is presented in the next section.

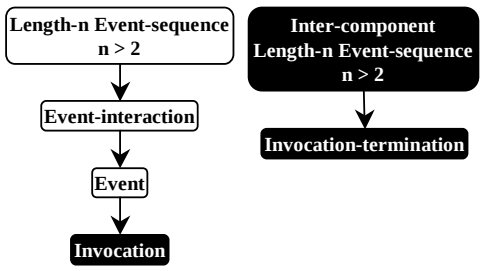

Figure 3: The Subsume Relation between Event-based Coverage Criteria. (Inter-component Criteria are shown in Reverse Color)

3.3 Inter-component Criteria

The goal of inter-component coverage criteria is to ensure that all interactions among components are tested. In GUIs, the interactions take the form of invocation of components, termination of components, and event-sequences that start with an event in one component and end with an event in another component.

3.3.1 Invocation Coverage

Intuitively, invocation coverage requires that each restricted-focus event in the GUI be performed at least once. Such a requirement is necessary to check whether each component can be invoked.

Definition: A set P of event-sequences satisfies the *invocation coverage criterion* if and only if for all restricted-focus events $i \in \mathcal{I}$, where $\mathcal{I}$ is the set of all restricted-focus events in the GUI, there is at least one event-sequence $p \in P$ such that event i is in p. □

Note that event coverage subsumes invocation coverage (Figure 3) since it requires that *all* events be performed at least once, including restricted-focus events.

3.3.2 Invocation-termination Coverage

It is important to check whether a component can be invoked and terminated.

Definition: The invocation-termination set $\mathcal{IT}$ of a GUI consists of all possible length 2 event sequences $< e_i, e_j >$, where e_i invokes component C_x and e_j terminates component C_x, for all components $C_x \in \mathcal{N}$. □

Intuitively, the invocation-termination coverage requires that all length 2 event sequences consisting of a restricted-focus event followed by one of the invoked component's termination event be tested.

Definition: A set P of event-sequences satisfies the *invocation-termination coverage criterion* if and only if for all $i \in \mathcal{IT}$, there is at least one event-sequence $p \in P$ such that i is in p. □

Satisfying the invocation-termination coverage criterion assures that each component is invoked at least once and then terminated immediately, if allowed by the GUI's specifications. For example, in WordPad, the component `FileOpen` is invoked by the event `Open` and terminated by either `Open` or `Cancel`. Note that WordPad's specifications do not allow `Open` to terminate the component unless a file has been selected. On the other hand, `Cancel` can always be used to terminate the component.

3.3.3 Inter-component Length-n Event-sequence Coverage

Finally, the inter-component length-n event-sequence coverage criterion requires testing all event-sequences that start with an event in one component and end with an event in another component. Note that such an event-sequence may use events from a number of components. A criterion is defined to cover all such interactions.

Definition: A set P of event-sequences satisfies the *inter-component length-n event-sequence coverage criterion* for components C_1 and C_2 if and only if P contains all length-n event-sequences $< e_1, e_2, e_3, ..., e_n >$ such that $e_1 \in Vertices(C_1)$ and $e_n \in Vertices(C_2)$. Events $e_2, e_3, ..., e_{n-1}$ may belong to C_1 or C_2 or any other component C_i. □

Note that the inter-component length-n event-sequence coverage subsumes invocation-termination coverage (Figure 3) since length-n event sequences also include length 2 sequences.

4. EVALUATING COVERAGE

Having formally presented intra- and inter-component coverage criteria, we now present algorithms to evaluate the coverage of a test suite using these criteria. In this section, we present algorithms to evaluate the coverage of the GUI for a given test suite. We show how to construct an event-flow graph and use it to evaluate intra-component coverage. Then we show how to construct an integration tree and use it to evaluate inter-component coverage.

4.1 Construction of Event-flow Graphs

The construction of event-flow graphs is based on the structure of the GUI. The classification of events in the previous section aids the automated construction of the event-flow graphs, which we describe next.

For each $\mathbf{v} \in \mathbf{V}$, we define `follow_set`($\mathbf{v}$) as the set of all events v_x such that v_x **follows** $\mathbf{v}$. Note that `follow_set`($\mathbf{v}$) is the set of outgoing edges in the event-flow graph. We determine `follow_set`($\mathbf{v}$) using the algorithm in Figure 4

```
ALGORITHM : GetFollows(
  v: Vertex or Event){                                          1
  IF EventType(v) = menu-open                                   2
   IF v ∈ B                                                     3
    return(MenuChoices(v) ∪ {v}) ∪ B)                           4
   ELSE                                                         5
    return(MenuChoices(v) ∪ {v}
      ∪ follow_set(parent(v)));                                 6
  IF EventType(v) = system-interaction                          7
   return(B);                                                   8
  IF EventType(v) = exit                                        9
   return(B of Invoking component);                             10
  IF EventType(v) = unrestricted-focus                          11
   return(B ∪ B of Invoked component);                          12
  IF EventType(v) = restricted-focus                            13
   return(B of Invoked component);                              14
}
```

Figure 4: Computing `follow_set`(v) for a Vertex v.

for each vertex $\mathbf{v}$. The recursive algorithm contains a switch structure that assigns `follow_set`($\mathbf{v}$) according to the type of each event. If the type of the event $\mathbf{v}$ is a menu-open event (`line 2`) and $\mathbf{v} \in \mathbf{B}$ (recall that $\mathbf{B}$ represents events that are available when a component is invoked) then the user may either perform $\mathbf{v}$ again, its sub-menu choices, or any event in $\mathbf{B}$ (`line 4`). However, if $\mathbf{v} \notin \mathbf{B}$ then the user may either perform all sub-menu choices of $\mathbf{v}$, $\mathbf{v}$ itself, or all events in `follow_set`(`parent`($\mathbf{v}$)) (`line 6`). We define `parent`($\mathbf{v}$) as any event that makes $\mathbf{v}$ available. If $\mathbf{v}$ is a system-interaction event, then after performing $\mathbf{v}$, the GUI reverts back to the events in $\mathbf{B}$ (`line 8`). If $\mathbf{v}$ is an exit event, i.e., an event that terminates a component, then `follow_set`($\mathbf{v}$) consists of all the top-level events of the invoking component (`line 10`). If the event type of $\mathbf{v}$ is an unrestricted-focus event then the available events are all top-level events of the invoked component available as well as all events of the invoking component (`line 12`). Lastly, if $\mathbf{v}$ is a restricted-focus event, then only the events of the invoked component are available.

4.2 Evaluating Intra-component Coverage

Having constructed an event-flow graph, we are now ready to evaluate the intra-component coverage of any given test suite using the elements of this graph. Figure 5 shows a dynamic programming algorithm to compute the percentage of length-n event-sequences tested. The final result of the algorithm is **Matrix**, where $\mathbf{Matrix}_{i,j}$ is the percentage of length-**j** event-sequences tested on component **i**.

The main algorithm is `ComputePercentageTested`. In this algorithm, two matrices are computed (`line 6,7`). $\mathbf{Count}_{i,j}$ is the number of length-**j** event-sequences in component **i** that have been covered by the test suite **T** (`line 6`). $\mathbf{Total}_{i,j}$ is the total number of all possible length-**j** event-sequences in component **i** (`line 7`). The subroutine `ComputeCounts` calculates the elements in **count** matrix. For each test case in **T**, `ComputeCounts` finds all possible event-sequences of different lengths (`line 19..21`). The subsequence $< t_k, \ldots, t_j >$ is obtained from the test case. Note that since `ComputeCounts` takes a union of the event sequences, there is no possibility of counting the same event sequence twice. The number of event-sequences of each length are counted in (`lines`

```
ALGORITHM : ComputePercentageTested(                          1
S: Set of Components;                                         2
T: Test Suite;                                                3
M: Maximum Event-sequence Length)                             4
{                                                             5
count ⟵ ComputeCounts(T, S, M);                               6
/* count_{i,j} is the tested number
of length-j event-sequences in component i */
total ⟵ ComputeTotals(S, M);                                  7
/* total_{i,j} is the total number
of length-j event-sequences in component i */
FOREACH i ∈ S DO                                              8
  FOR j ⟵ 1 TO M DO                                           9
    Matrix_{i,j} ⟵ (count_{i,j}/total_{i,j}) × 100;           10
return(Matrix)}                                               11

SUBROUTINE : ComputeCounts(                                   12
T: Test Suite; S: Set of Components;                          13
M: Maximum Event-sequence Length)                             14
{                                                             15
FOREACH i ∈ S DO                                              16
  A ⟵ {}; /* Empty Set */                                     17
  FOREACH t ∈ T DO                                            18
    FOR k ⟵ 1 TO |t| DO                                       19
      FOR j ⟵ k TO |t| DO                                     20
        A ⟵ A ∪ {< t_k...t_j >}                               21
  FOR j ⟵ 1 TO M DO                                           22
    /* count number of sets of length j */
    count_{i,j} ⟵ NumberOfSetsOfLength(S, j);                 23
return(count)}                                                24

SUBROUTINE : ComputeTotals(                                   25
S: Set of Components;                                         26
M: Maximum Event-sequence Length)                             27
{FOREACH j ∈ S DO                                             28
  E ⟵ Edges(j);                                               29
  V ⟵ Vertices(j);                                            30
  FOREACH i ∈ V DO                                            31
    freq_i ⟵ 1;                                               32
  total_{1,j} ⟵ |V|;                                          33
  FOREACH i ∈ V DO                                            34
    newfreq_i ⟵ 0;                                            35
  FOR k ⟵ 2 TO M DO                                           36
    FOREACH i ∈ V DO                                          37
      x ⟵ follow_set(i);                                      38
      total_{j,k} ⟵ total_{j,k} + |x| × freq_i;               39
      FOREACH l ∈ x DO                                        40
        newfreq_j + +;                                        41
    freq ⟵ newfreq;                                           42
    FOREACH i ∈ V DO                                          43
      newfreq_i ⟵ 0;                                          44
return(total)}                                                45
```

Figure 5: Computing Percentage of Tested Length-n Event-sequences of All Components.

22, 23). Intuitively, the ComputeTotals subroutine starts with single-length event-sequences, i.e., individual events in the GUI (lines 31..33). Using follow_set (line 38), the event-sequences are lengthened one event at each step. A counter keeps track of the number of event-sequences created (line 39). For every element in the follow_set of **i**, the frequency counter **newfreq** is incremented (lines 40..41), hence counting the total number of outgoing edges in the event-flow graph.

The result of the algorithm is **Matrix**, the entries of which can be interpreted as follows:

Event Coverage requires that individual events in the GUI be exercised. These individual events correspond to length 1 event-sequences in the GUI. $\mathbf{Matrix}_{j,1}$, where $j \in S$, represents the percentage of individual events covered in each component.

Event-interaction Coverage requires that all the edges of the event-flow graph be covered by at least one test case. Each edge is effectively captured as a length 2 event-sequence. $\mathbf{Matrix}_{j,2}$, where $j \in S$, represents the percentage of branches covered in each component **j**.

Length-n Event-sequence Coverage is available directly from **Matrix**. Each column **i** of **Matrix** represents the number of length-**i** event-sequences in the GUI.

4.3 Evaluating Inter-component Coverage

Once all the components in the GUI have been identified, the integration tree is constructed by adding, for each restricted-focus event e_x, the element (C_x, C_y) to $\mathcal{B}$ where C_x is the component that contains e_x and C_y is the component that it invokes. The integration tree is used in various ways to identify interactions among components. For example, in Figure 2 a subset of all possible pairs of components that interact would be { (Main, FileNew), (Main, FileOpen), (Main, Print), (Main, FormatFont), and (Print, Properties) }. To identify sequences such as the ones from Main to Properties, we traverse the integration tree in a bottom-up manner, identifying interactions among Print and Properties. We then merge Print and Properties to form a *super-component* called PrintProperties, and check interactions among Main and PrintProperties. This process continues until all components have been merged into a single super-component.

Evaluating the inter-component coverage of a given test suite requires computing the (1) invocation coverage, (2) invocation-termination coverage, and (3) length-n event sequence coverage. The total number of length 1 event sequences required to satisfy the invocation coverage criterion is equal to the number of restricted-focus events available in the GUI. The percentage of restricted-focus events actually covered by the test cases is $(x/\mathcal{I}) \times 100$, where x is the number of restricted-focus events in the test cases, and $\mathcal{I}$ is the total number of restricted-focus events available in the GUI. Similarly, the total number of length 2 event sequences required to satisfy the invocation-termination criterion is $\sum(I_i \times T_i)$, where I_i and T_i are the number of restricted-focus and termination events that invoke and terminate component C_i respectively. The percentage of invocation-termination pairs actually covered by the test cases is $(x/\sum(I_i \times T_i)) \times 100$, where x is the number of invocation-termination pairs in the test cases.

Computing the percentage of length-n event sequences is slightly more complicated. The algorithm shown in Figure 6 computes the percentage of length-n event sequences tested among GUI components. Intuitively, the algorithm

```
ALGORITHM : Integrate(                                          1
T: Integration Tree)                                            2
{                                                               3
IF Leaf(T)                                                      4
  return(T);                                                    5
newT ⟵ T;                                                       6
FORALL c ∈ Children(T) DO                                       7
  Integrate(c);                                                 8
  ComputeTotalInteractions(newT, c);                            9
  Matrix_{newT+c} ⟵ TestedEventSeq_{newT+c}/Total;              10
}                                                               11

SUBROUTINE : ComputeTotalInteractions(                          12
C_1: Component 1;                                               13
C_2: Component 2)                                               14
{                                                               15
FOR i ⟵ 1 TO M DO                                               16
  Total_i ⟵ 0;                                                  17
x ⟵ GetCallingEvent(C_1, C_2);                                  18
FOR i ⟵ 1 TO M DO                                               19
  /* get freq table of C_1 for event-seq of length i */        20
  F_1 ⟵ GetFreqTable(C_1, i);                                   21
  /* Add all values in column x */                              22
  p ⟵ addColumn(x, F_1);                                        23
  FOR j ⟵ 1 TO M DO                                             24
  /* get freq table of C_2 for event-seq of length j */         25
   F_2 ⟵ GetFreqTable(C_2, j);                                  26
   q ⟵ 0;                                                       27
   FOREACH k ∈ B of C_2 DO                                      28
    q ⟵ q + addRow(k, F_2);                                     29
   Total_{i+j} ⟵ Total_{i+j} + p × q;                           30
 ComputeFreqMatrix(C_1, C_2);                                   31
return(Total);                                                  32
}
```

Figure 6: Computing Percentage of Tested Length-n Event-sequences of All Components.

obtains the number of event sequences that end at a certain restricted-focus event. It then counts the number of event sequences that can be extended from these sequences into the invoked component. The main algorithm called `Integrate` is recursive and performs a bottom-up traversal of the integration tree **T** (`line 2`). Other than the recursive call (`line 8`), `Integrate` makes a call to `ComputeTotalInteractions` that takes two components as parameters (`lines 13,14`). It initializes the vector **Total** for all path lengths **i** ($1 \leq \mathbf{i} \leq \mathbf{M}$) (`line 16,17`). We assume that a matrix (**freq**) has been stored for each component. The **freq** matrix is similar to the **freq** vector already computed in the algorithm in Figure 5. $\mathbf{freq}_{i,j}$ is the number of event-sequences that start with event i and end with event j. After obtaining both frequency matrices for both $\mathbf{C}_1$ and $\mathbf{C}_2$, for all path lengths (`lines 21,26`), the new vector **Total** is obtained by adding the frequency entries from $\mathbf{F}_1$ and $\mathbf{F}_2$ (`lines 28..30`). A new frequency matrix is computed for the super-component "$\mathbf{C}_1\mathbf{C}_2$" (`line 31`). This new frequency matrix will be utilized by the same algorithm to integrate "$\mathbf{C}_1\mathbf{C}_2$" to other components.

The results of the above algorithm are summarized in **Matrix**. $\mathbf{Matrix}_{i,j}$ is the percentage of length-**j** event-sequences that have been tested in the super-component represented by the label **i**.

5. CASE STUDY

We performed a case study on our version of WordPad to determine the (1) total number of event sequences required to test the GUI and hence enable a test designer to compute the percentage of event sequences tested, (2) correlation between event-based coverage of the GUI and statement coverage of the underlying code, and (3) time taken to evaluate the coverage of a given test suite and usefulness of the coverage report to guide further testing.

In the case study, we employed our own specifications and implementation of the WordPad software. The software consists of 36 modal windows, and 362 events (not counting short-cuts). Our implementation of WordPad is similar to Microsoft's WordPad except for the **Help** menu, which we did not model.

5.1 Computing Total Number of Event-sequences for WordPad

In this case study, we wanted to determine the total number of event sequences that our new criteria specify to test parts of WordPad. We performed the following steps:

Identifying Components and Events: Individual WordPad components and events within each component were identified. Table 1 presented earlier shows some of the components of WordPad that we used in our case study.

Creating Event-flow Graphs: The next step was to construct an event-flow graph for each component. In Figure 1 we showed a part of the event-flow graph of the most important component, `Main`. Recall that each node in the event-flow graph represents an event.

Computing Event-sequences: Once the event-flow graphs were available, we computed the total number of possible event-sequences of different lengths in each component by using the `computeTotals` subroutine in Figure 5. Note that these event-sequences may also include infeasible event-sequences. The total number of event-sequences is shown in Table 2. The rows represent the components and the shaded rows represent the inter-component interactions. The columns represent different event-sequence lengths. Recall that an event-sequence of length 1 represents *event coverage* whereas an event-sequence of length 2 represents *event-interaction coverage*. The columns 1' and 2' represent invocation and invocation-termination coverage respectively.

The results of this case study show, not surprisingly, that the total number of event sequences grows with increasing length. Note that longer sequences subsume shorter sequences; e.g., if all event sequences of length 5 are tested, then so are all sequences of length-i, where $i \leq 4$. It is difficult to determine the maximum length of event sequences needed to test a GUI. The large number of event sequences show that it is impractical to test a GUI for all possible event sequences. Rather, depending on the resources, a subset of "important" event sequences should be identified, generated

Component Name	Event-sequence Length							
	1'	2'	1	2	3	4	5	6
Main			56	791	14354	255720	4490626	78385288
FileOpen			10	80	640	5120	40960	327680
FileSave			10	80	640	5120	40960	327680
Print			12	108	972	8748	78732	708588
Properties			13	143	1573	17303	190333	2093663
PageSetup			11	88	704	5632	45056	360448
FormatFont			9	63	441	3087	21609	151263
Print+Properties	1	2		13	260	3913	52520	663013
Main+FileOpen	1	2		10	100	1180	17160	278760
Main+FileSave	1	2		10	100	1180	17160	278760
Main+PageSetup	1	2		11	110	1298	18876	306636
Main+FormatFont	1	2		9	81	909	13311	220509
Main+Print+Properties				12	145	1930	28987	466578

Table 2: Total Number of Event-sequences for Selected Components of WordPad. Shaded Rows Show Number of Interactions Among Components.

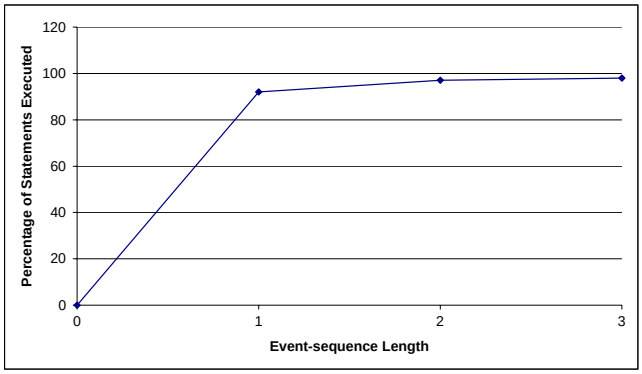

Figure 7: The Correlation Between Event-based Coverage and Statement Coverage of WordPad.

and executed. Identifying such important sequences requires that they be ordered by assigning a *priority* to each event sequence. For example, event sequences that are performed in the `Main` component may be given higher priority since they may be used more frequently; all the users start interacting with the GUI using the `Main` component. The components that are deepest in the integration tree may be used the least. This observation leads to a heuristic for ordering the testing of event sequences within components of the GUI. The structure of the integration tree may be used to assign priorities to components; `Main` will have the highest priority, decreasing for components at the second level, with the deepest components having the lowest priority. A large number of event sequences in the high priority components may be tested first; the number will decrease for low priority components.

5.2 Correlation Between Event-based Coverage and Statement Coverage

In this case study, we wanted to determine exactly which percentage of the underlying code is executed when event-sequences of increasing length are executed on the GUI. We wanted to see how code coverage relates to event coverage. We performed the following steps:

Code Instrumentation: We instrumented the underlying code of WordPad to produce a *statement trace*, i.e., a sequence of statements in the order in which they are executed. Examining such a trace allowed us to determine which statements are executed by a test case.

Event-sequence Generation: We wanted to generate all event-sequences up to a specific length. We modified `ComputeTotals` in Figure 5 to produce an event-sequence generation algorithm that constructs event sequences of increasing length. The dynamic programming algorithm constructs all event sequences of length 1. It then uses `follow_set` to extend each event sequence by one event, hence creating all length 2 event-sequences. We generated all event-sequences up to length 3. In all we obtained 21659 event-sequences.

Controlling GUI's State: Bringing a software to a state S_i in which a test case T_i may be executed on it is traditionally known as the *controllability problem* [1]. This problem also occurs in GUIs and for each test case, appropriate events may need to be performed on the GUI to bring it to the state S_i. We call this sequence of events the *prefix*, P_i, of the test case. Although generating the prefix in general may require the development of expensive solutions, we used the following heuristic for this study: we executed each test case in a fixed state S_0 in which WordPad contains text, part of the text was highlighted, the clipboard contains a text object, and the file system contains two text files. We traversed the event-flow graphs and the integration tree to produce the prefix of each test case. We do, however, note that using this heuristic may render some of the event sequences non-executable because of infeasibility. We will later see that such sequences do exist but are of no consequence to the results of this study. We have modified WordPad so that no statement trace is produced for P_i.

Test-case Execution: After all event-sequences up to length 3 were obtained, we executed them on the GUI using our automated test executor [10] and obtained all the execution traces. The test case executor executed without any intervention for 30 hours. We note that 4189 (or 19.3%) of the test cases could not be executed because of infeasibility.

Analysis: In analyzing the traces for our study, we determined the new statements executed by event-sequences of length 1, i.e., individual events. The graph in Figure 7 shows that almost 92% of the statements were executed by just these individual events. As the length of the event sequences increases, very few new statements are executed (5%). Hence, a high statement coverage of the underlying code may be obtained by executing short event sequences.

The relationship between event sequences and code, obtained from this case study, can be explained in terms of the design of the WordPad GUI. Since the GUI is an event-driven software, a method called an event handler is implemented for each event. Executing an event caused the execution of its corresponding event handler. Code inspection of the WordPad implementation revealed that there were few or no branch statements in the code of the event handler. Consequently, when an event was performed, most of the statements in the event-handler were executed. Hence

high statement coverage was obtained by just performing individual events. Whether other GUIs exhibit similar behavior requires a detailed analysis of a number of GUIs and their underlying code.

The result shows that statement coverage of the underlying code can be a misleading coverage criterion for GUI testing. A test designer who relies on statement coverage of the underlying code for GUI testing may test only short event sequences. However, testing only short sequences is not enough. Longer event sequences lead to different states of the GUI and testing these sequences may help detect a larger number of faults than testing short event sequences. For example, in WordPad, the event `Find Next` (obtained by clicking on the `Edit` menu) can only be executed after at least 6 events have been performed; the shortest sequence of events needed to execute `Find Next` is <`Edit, Find, TypeInText, FindNext2, OK, Edit, Find Next`>, which has 7 events. If only short sequences (< 3) are executed on the GUI, a bug in `Find Next` may not be detected. Extensive studies of the fault-detection capabilities of executing short and long event sequences for GUI testing are needed, and are targeted for future work. Another possible extension to this study is to determine the correlation between event-based coverage and other code-based coverage, e.g., branch coverage.

5.3 Evaluating the Coverage of a Test Suite

We also wanted to determine the time taken to evaluate the coverage of a given test suite and how the resulting coverage report could guide further testing. We used our previously developed planning-based test case generation system called Planning Assisted Tester for grapHical user interface Systems(PATHS) to generate test cases [8]. We performed the following steps:

Identifying Tasks: In PATHS, commonly used *tasks* were identified. A task is an activity to be performed by using the events in the GUI. In PATHS, the test designer inputs tasks as pairs ($\mathcal{I}$, $\mathcal{G}$), where $\mathcal{I}$ is the initial GUI state before the task is performed and $\mathcal{G}$ is the final GUI state after the task has been performed. We carefully identified 72 different tasks, making sure that each task exercised at least one unique feature of WordPad. For example, in one task we modified the font of text, in another we printed the document on A4 size paper.

Generating Test Cases: Multiple test cases were generated using a plan generation system to achieve these tasks. In this manner, we generated 500 test cases (multiple cases for each task).

Coverage Evaluation: After the test cases were available, we executed the algorithms of Figures 5 and 6. The algorithms were implemented using Perl and *Mathematica* [19] and were executed on a Sun UltraSPARC workstation (Sparc Ultra 4) running SunOS 5.5.1. Even with the inefficiencies inherent in the Perl and Mathematica implementation, we could process the 500 test cases in 47 minutes (clock time). The results of applying the algorithms are summarized as coverage reports in Tables 3 and 4. Table 3 shows the actual number of event-sequences that the test cases covered. Table 4 presents the same data, but as a percentage of the total number of event-sequences. Column 1 in Table 4 shows close to 90% event coverage. The remaining 10% of the events (such as `Cancel`) were never used by the planner since they did not contribute to a goal. Column 2 shows the event-interaction coverage and the test cases achieved 40-55% coverage. Note that since all the components were invoked at least once, 100% invocation coverage (column 1') was obtained. However, none of the components were terminated immediately after being invoked. Hence, no invocation-termination coverage (column 2') was obtained.

	Event-sequence Length							
Component Name	**1'**	**2'**	**1**	**2**	**3**	**4**	**5**	**6**
Main			49	321	1567	915	1231	1987
FileOpen			9	45	112	37	23	179
FileSave			9	33	132	65	193	67
Print			11	37	313	787	3085	1314
Properties			12	65	434	312	1848	1235
PageSetup			10	43	179	144	298	233
FormatFont			8	23	172	422	142	84
Print+Properties	1	0		6	133	320	2032	326
Main+FileOpen	1	0		4	11	120	223	453
Main+FileSave	1	0		2	13	102	217	769
Main+PageSetup	1	0		5	67	56	367	233
Main+FormatFont	1	0		3	23	47	129	227
Main+Print+Properties				6	56	123	189	423

Table 3: The Number of Event-sequences for Selected Components of WordPad Covered by the Test Cases.

	Event-sequence Length							
Component Name	**1'**	**2'**	**1**	**2**	**3**	**4**	**5**	**6**
Main			88	41	10.92	0.36	0.03	0.00
FileOpen			90	56	17.50	0.72	0.06	0.05
FileSave			90	41	20.63	1.27	0.47	0.02
Print			92	34	32.20	9.00	3.92	0.19
Properties			92	45	27.59	1.80	0.97	0.06
PageSetup			91	49	25.43	2.56	0.66	0.06
FormatFont			89	37	39.00	13.67	0.66	0.06
Print+Properties	100	0		46	51.15	8.18	3.87	0.05
Main+FileOpen	100	0		40	11.00	10.17	1.30	0.16
Main+FileSave	100	0		20	13.00	8.64	1.26	0.28
Main+PageSetup	100	0		45	60.91	4.31	1.94	0.08
Main+FormatFont	100	0		33	28.40	5.17	0.97	0.10
Main+Print+Properties				50	38.62	6.37	0.65	0.09

Table 4: The Percentage of Total Event-sequences for Selected Components of WordPad Covered by the Test Cases.

This result shows that the time taken to evaluate the coverage of a large test suite is reasonable. Looking at columns 4, 5, and 6 of Table 4, we note that only a small percentage of length 4, 5, and 6 event sequences were tested. The test designer can evaluate the importance of testing these longer sequences and perform additional testing. Also, the two-dimensional structure of Table 4 helps target specific components and component-interactions. For example, 60% of of length 2 interactions among `Main` and `PageSetup` have been tested whereas only 11% of the interactions among `Main` and `FileOpen` have been tested. Depending on the rel-

ative importance of these components and their interactions, the test designer can focus on testing these specific parts of the GUI.

The coverage report produced from this case study shows two important weaknesses of PATHS. First, PATHS did not use events such as `Cancel` since they did not contribute to the planning goal, resulting in loss of coverage as seen in column 1 of Table 4. Second, PATHS did not generate event sequences that invoke a component and terminate it immediately since such preemptive termination did not contribute to the final goal. This behavior of the planning-based test-case generator resulted in loss of coverage as seen in column 2' of Table 4. Note that, in practice, GUI users can, and do terminate components without interacting with other events in the component. It is important to test the GUI for such event sequences, perhaps by employing other testing techniques. An important lesson demonstrated from this case study is that it is necessary to combine several techniques to test a GUI software, so that weaknesses of one technique do not have too much impact on the overall testing results. Rather, the combined strengths of several testing techniques will result in better testing of the GUI software.

6. RELATED WORK

Very little research has been reported on developing coverage criteria for GUIs. The main exception is the work by Ostrand et al. who briefly indicate that a *model-based method* may be useful for improving the coverage of a test suite [12]. However, they have deferred a detailed study of the coverage of the generated test cases using this type of GUI model to future work.

There is a close relationship between test-case generation techniques and the underlying coverage criteria used. Much of the literature on GUI test case generation focuses on describing the algorithms used to generate the test cases [14, 18, 6]. Little or no discussion about the underlying coverage criteria is presented. In the next few paragraphs, we present a discussion of some of the methods used to develop test cases for GUIs and their underlying coverage criteria. We also present a discussion of automated test case generation techniques that offer a unique perspective of GUI coverage.

The most commonly available tools to aid the test designer in the GUI testing process include record/playback tools [15, 3]. These tools record the user events and GUI screens during an interactive session. The recorded sessions are later played back whenever it is necessary to generate the same GUI events. Record/playback tools provide no functionality to evaluate the coverage of a test suite. The primary reason for no coverage support is that these tools lack a global view of the GUI. The test cases are constructed individually with a local perspective. Several attempts have been made to provide more sophisticated tools for GUI testing. One popular technique is programming the test case generator [7]. The test designer develops programs to generate test cases for a GUI. The use of loops, conditionals, and data selection switches in the test case generation program gives the test designer a broader view of the generated test cases' coverage.

Several finite-state machine (FSM) models have also been proposed to generate test cases [14]. Once an FSM is built, coverage of a test suite is evaluated by the number of states visited by the test case. This method of evaluating coverage of a test suite needs to be studied further as an accurate representation of the GUI's navigation results in an infinite number of states.

White et al. presents a new test case generation technique for GUIs [18]. The test designer/expert manually identifies a responsibility, i.e., a GUI activity. For each responsibility, a machine model called the complete interaction sequence (CIS) is identified manually. To reduce the size of the test suite, the CIS is reduced using constructs/patterns in the CIS. The example presented therein showed that testing could be performed by using 8 test cases instead of 48. However, there is no discussion of why no loss of coverage will occur during this reduction. Moreover, further loss of coverage may occur in identifying responsibilities and creating the CIS. The merit of the technique will perhaps be clearer when interactions between the CIS are investigated.

7. CONCLUSION

In this paper, we present new coverage criteria for GUI testing based on GUI events and their interactions. A unit of testing called a GUI component is defined. We identify the events within each component and represented them as an event-flow graph. Three new coverage criteria for events within a component are defined: event, event-interaction, and length-n event-sequence coverage. We define an integration tree to identify events among components, and three inter-component coverage criteria: invocation, invocation-termination and inter-component length-n event-sequence coverage.

In the future we plan to examine the effects of the GUI's structure on its testability. As GUIs become more structured, the integration tree becomes more complex and inter-component testing becomes more important.

We also plan to explore the possibility of using the event-based coverage criteria for software other than GUIs. We foresee the use of these criteria for (1) object-oriented software, which use messages/events for communication among objects, (2) networking software, which use messages for communication, and (3) the broader class of reactive software, which responds to events.

8. REFERENCES

[1] D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker. A framework for testing database applications. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis (ISSTA)*, pages 147–157, 2000.

[2] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, 9(6):686–709, Nov. 1983.

[3] M. L. Hammontree, J. J. Hendrickson, and B. W. Hensley. Integrated data capture and analysis tools for research and testing an graphical user interfaces. In *Proceedings of the Conference on Human Factors in*

Computing Systems, pages 431–432, New York, NY, USA, May 1992. ACM Press.

[4] M. J. Harrold and M. L. Soffa. Interprocedual data flow testing. In R. A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Testing, Analysis, and Verification (TAV3)*, pages 158–167, 1989.

[5] P. C. Jorgensen and C. Erickson. Object-oriented integration testing. *Communications of the ACM*, 37(9):30–38, Sept. 1994.

[6] D. J. Kasik and H. G. George. Toward automatic generation of novice user test scripts. In *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*, pages 244–251, New York, 13–18 Apr. 1996. ACM Press.

[7] L. R. Kepple. The black art of GUI testing. *Dr. Dobb's Journal of Software Tools*, 19(2):40, Feb. 1994.

[8] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 257–266. ACM Press, May 1999.

[9] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In D. S. Rosenblum, editor, *Proceedings of the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-00)*, pages 30–39, NY, Nov. 8–10 2000. ACM Press.

[10] A. M. Memon, M. E. Pollack, and M. L. Soffa. A planning-based approach to GUI testing. In *Proceedings of The 13th International Software/Internet Quality Week*, May 2000.

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

[12] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for GUI systems. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)*, pages 82–92, New York, Mar.2–5 1998. ACM Press.

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

[14] R. K. Shehady and D. P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 80–88, Washington - Brussels - Tokyo, June 1997. IEEE Press.

[15] L. The. Stress Tests For GUI Programs. *Datamation*, 38(18):37, Sept. 1992.

[16] E. J. Weyuker. The applicability of program schema results to programs. *International Journal of Computer and Information Sciences*, 8(5):387–403, Oct. 1979.

[17] E. J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8(4):587–598, 1979.

[18] L. White and H. Almezen. Generating test cases for GUI responsibilities using complete interaction sequences. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 110–121, Oct. 8–11 2000.

[19] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, Reading, Massachusetts, 1988.

[20] H. Zhu and P. Hall. Test data adequacy measurements. *Software Engineering Journal*, 8(1):21–30, Jan. 1993.