GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing

Atif Memon
Department of Computer Science
and Fraunhofer Center for
Experimental Software Engineering
University of Maryland
College Park, Maryland, USA
atif@cs.umd.edu

Ishan Banerjee, Adithya Nagarajan
Department of Computer Science
University of Maryland
College Park, Maryland, USA
{ishan, sadithya}@cs.umd.edu

Abstract

Graphical user interfaces (GUIs) are important parts of today's software and their correct execution is required to ensure the correctness of the overall software. A popular technique to detect defects in GUIs is to test them by executing test cases and checking the execution results. Test cases may either be created manually or generated automatically from a model of the GUI. While manual testing is unacceptably slow for many applications, our experience with GUI testing has shown that creating a model that can be used for automated test case generation is difficult.

We describe a new approach to reverse engineer a model represented as structures called a GUI forest, event-flow graphs and an integration tree directly from the executable GUI. We describe "GUI Ripping", a dynamic process in which the software's GUI is automatically "traversed" by opening all its windows and extracting all their widgets (GUI objects), properties, and values. The extracted information is then verified by the test designer and used to automatically generate test cases. We present algorithms for the ripping process and describe their implementation in a tool suite that operates on Java and Microsoft Windows' GUIs.

We present results of case studies which show that our approach requires very little human intervention and is especially useful for regression testing of software that is modified frequently. We have successfully used the "GUI Ripper" in several large experiments and have made it available as a downloadable tool.

1 Introduction

Graphical user interfaces (GUIs) are one of the most important parts of today's software [13]. They make software easy to use by providing the user with highly visual controls that represent everyday objects such as menus, buttons, lists, and windows. Recognizing the importance of GUIs, software developers are dedicating large parts of the code to implementing GUIs [12]. The correctness of this code is essential to the correct execution of the overall software. A popular technique to detect defects in software is *testing* [3, 2, 23]. During testing, *test cases* are created and executed on the software. Test cases may either be created manually by a tester [10, 27, 8] or automatically by using a model of the software derived from its specifications [20]. In all our work to date [20, 17, 21, 16, 18, 19, 15, 12, 14], we have observed that software specifications are rarely in a form to be used for automated GUI testing.

GUI testing requires that test cases (sequences of GUI *events* that exercise GUI *widgets*) be generated and executed on the GUI [13]. However, currently available techniques for obtaining GUI test cases are resource intensive, requiring significant human intervention. The most popular technique to test GUIs is by using *capture/replay tools* [10]. When using a capture/replay tool, a human tester interacts with the *application under test* (AUT); the capture component of the tool stores this interaction in a file that can be replayed later using the replay component of the tool. Our experience has shown that generating a typical test case with 50 events for different widgets takes 20-30 minutes using capture-replay tools.

A few automated GUI test case generation techniques have been proposed [20]. However, they all require creating a model of the GUI – a significant resource intensive step that intimidates many practitioners and prevents the application of the techniques. In this paper, we present a technique, called *GUI Ripping* to reverse engineer the GUI's model directly from the *executing* GUI. Once verified by the test designer, this model is then used to automatically generate test cases. GUI ripping has numerous other applications such as reverse engineering of COTS GUI products to test them within the context of their use, porting and controlling legacy applications to new platforms [22], and developing model checking tools for GUIs [6]. For space reasons, in this paper, we will provide details relevant to the testing process.

GUI ripping is a dynamic process that is applied to an executing software's GUI. Starting from the software's first window (or set of windows), the GUI is "traversed" by opening all child windows. All the window's *widgets* (building blocks of the GUI, e.g., buttons, text-boxes), their *properties* (e.g., background-color, font), and *values* (e.g., red, Times New Roman, 18pt) are extracted. Developing this process has several challenges that required us to develop novel solutions. First, the source code of the software may not always be available; we had to develop techniques to extract information from the executable files. Second, there are no GUI standards across different platforms and implementations; we had to extract all the information via low-level implementation-dependent system calls, which we have found are not always well-documented. Third, some implementations may provide less information than necessary to perform automated testing; we had to rely on heuristics and human intervention to determine missing parts. Finally, the presence of *infeasible paths* in GUIs prevents full automation. For example, some windows may be available only after a valid password has been provided. Since the GUI Ripper may not have access to the password, it may not be able to extract information from such windows. We had to provide another process and tool support to visually add parts to the extracted GUI model.

We use GUI ripping to extract both the structure and execution behavior of the GUI – both essential for automated testing. We represent the GUI's structure as a *GUI forest* and its execution behavior as *event-flow graphs*, and an *integration tree* [21]. Each node of the GUI forest represents a window and encapsulates all the widgets, properties and values in that window; there is an edge from node x to node y if the window represented by y is opened by performing an event in the window represented by node x, e.g., by clicking on a button. Intuitively, event-flow graphs and the integration tree show the *flow of events* in the GUI. We provide details of these structures in Section 2.

We have implemented our algorithm in a software called the *GUI Ripper*. We use the GUI Ripper as a central part of two large software systems called GUITAR[1] and DART (Daily Automated Regression Tester) to generate, execute, verify GUI test cases, and perform regression testing [15]. We provide details of two instances of the GUI Ripper, one for Microsoft Windows and the other for Java Swing applications. We then empirically evaluate the performance of the ripper on four Java applications with complex GUIs, Microsoft's WordPad, Yahoo Messenger, and Winzip. The results of our empirical studies show that the ripping process is efficient, in that it is very fast and requires little human intervention. We also show that relative to other testing activities, ripping consumes very little resources. We also observe that automated testing would not be possible without the help of the GUI Ripper.

The specific contributions of our work include the following.

- We provide an efficient algorithm to extract a software's GUI model without the need for its source code.
- We describe a new structure called a *GUI forest*.
- We provide implementation details of a new tool that can be applied to a large number of MS Windows and Java Swing GUIs.

In the next section, we present a formal model of the GUI specifications that are obtained by the GUI Ripper. In Section 3, we present the design of the ripper and provide an algorithm that can be used to implement the ripper. In Section 4 we discuss the MS Windows and Java implementations of the GUI Ripper. In Section 5, we empirically evaluate our algorithms on several large and popular software. We then conclude with a discussion of related work in Section 6, and ongoing and future work in Section 7.

2 GUI Model

During GUI ripping, a representation of the GUI that models its structure and execution behavior is created from the executing GUI. In this section, we describe this representation and formally describe the models used for testing. Since developing general reverse engineering solutions for all types of GUIs is difficult, we focus on an important subclass of GUIs described next.

2.1 What is a "GUI"?

GUIs, by their very nature, are hierarchical. This hierarchy is reflected in the grouping of events in windows, dialogs, and hierarchical menus. A typical GUI user focuses on events related by their functionality by opening a particular window or clicking on a pull-down menu. 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 widgets they contain, and the properties (attributes) of those widgets. Formally, the class of GUIs of interest may be defined as follows:

Definition: A *G̲raphical U̲ser I̲nterface (GUI)* is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events, from a fixed set of events and produces deterministic graphical output. A GUI contains graphical *widgets*; each widget

[1]`http://guitar.cs.umd.edu`

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 widgets 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 focuses on techniques to reverse engineer the class of GUIs defined above.

2.2 GUI Forest

The first GUI representation that we obtain during the ripping process is called the GUI forest. Intuitively, the GUI forest represents the structure of the GUI's windows (as nodes of the forest), and the hierarchical relationship between windows (as edges). Each node encapsulates the state of a window that constitutes the window's widgets, their properties, and values.

We model a GUI window as a set of *widgets* (e.g., buttons, labels, text fields) that constitute the window, a set of *properties* (e.g., background color, size, font) of these widgets, and a set of *values* (e.g., red, bold, 16pt) associated with the properties. Each window will contain certain types of widgets with associated properties. At any point during its execution, the window can be described in terms of the specific widgets that it currently contains and the values of their properties. More formally, we model a window at a particular time t in terms of:

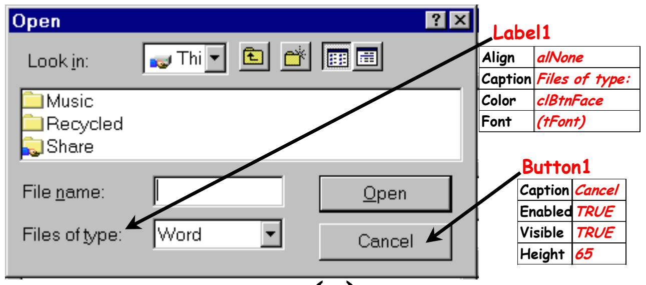
- *widgets* $W = \{w_1, w_2, ..., w_l\}$, i.e., the widgets that the window currently contains,
- *properties* $P = \{p_1, p_2 ..., p_m\}$ of the widgets, and
- *values* $V = \{v_1, v_2 ..., v_n\}$ of the properties.

For example, consider the `Open` window shown in Figure 1(a). This window contains several widgets, two of which are explicitly labeled, namely `Button1` and `Label1`; for each, a small subset of properties is shown. Note that all widget types have a designated set of properties and all properties can take values from a designated set.

The set of widgets and their properties can be used to create a model of the *state* of the window.

Definition: The *state* of a window at a particular time t is the set S of triples $\{(w_i, p_j, v_k)\}$, where $w_i \in W$, $p_j \in P$, and $v_k \in V$. □

A description of the *complete state* would contain information about the types of *all* the widgets currently extant in



Align	alNone
Caption	Files of type:
Color	clBtnFace
Font	(tFont)

Caption	Cancel
Enabled	TRUE
Visible	TRUE
Height	65

(a)

State **= {(Label1, Align, alNone), (Label1, Caption, "Files of type:"), (Label1, Color, clBtnFace), (Label1, Font, (tfont)), (Form1, WState, wsNormal), (Form1, Width, 1088), (Form1, Scroll, TRUE), (Button1, Caption, Cancel), (Button1, Enabled, TRUE), (Button1, Visible, TRUE), (Button1, Height, 65), …}**

(b)

Figure 1. (a) `Open` **window, (b) its Partial State**

the window, as well as *all* of the properties and their values for each of those widgets. The state of the `Open` window, partially shown in Figure 1(b), contains all the properties of all the widgets in `Open`.

The windows of the GUI form a hierarchy – once the software is invoked, the user is presented with a *top-level* window (or set of windows). All other windows of the GUI are invoked from one of the top-level windows or from their descendents. In general, the relationships among windows may be represented by a set of directed acyclic graphs (DAGs), since multiple windows may invoke a window. However, each DAG can be converted into a tree by copying nodes. A tree model simplifies our algorithms based on tree traversals. Note that since most GUIs have a single top-level window, in most cases, the forest reduces to a single tree. Formally, we define a GUI forest as:

Definition: A *GUI forest* is a triple $< \mathcal{W}, \mathcal{T}, \mathcal{E} >$, where $\mathcal{W}$ is the set of windows in the GUI and $\mathcal{T} \subseteq \mathcal{W}$ is a designated set of windows called the `top-level` windows. $\mathcal{E}$ is the set of directed edges: there is an edge from node x to node y if the window represented by y is opened by performing an event in the window represented by node x. □

Different types of GUI forests may be obtained depending on the types of windows that the GUI contains. For the purpose of testing, we distinguish between two different types of windows: *modal* windows and *modeless* windows.

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 is an example of a modal window in MS Word – when the user performs the event

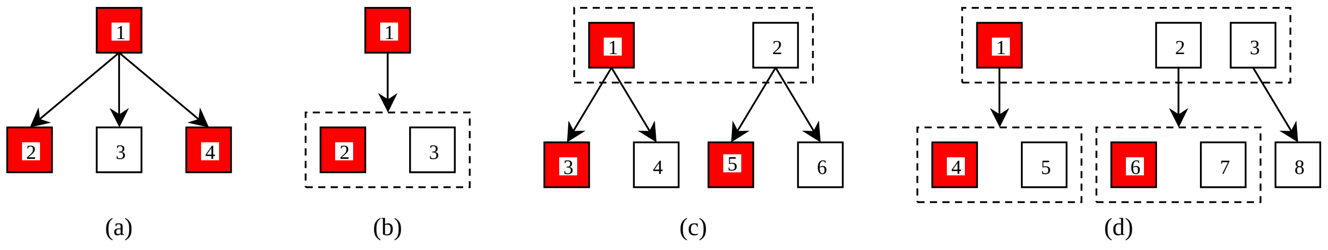


Figure 2. Examples of GUI Forests.

`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`.

Figure 2 shows some examples of GUI forests. The shaded nodes represent modal windows and unshaded nodes represent modeless windows. Dashed boxes group windows that open simultaneously. Figure 2(a) shows the simplest case of a GUI in which window 1 is a modal window; three events in window 1 are used to open three windows 2, 3 and 4, where 2 and 4 are modal, and 3 is modeless. Figure 2(b) shows a more complex case of a GUI in which window 1 contains an event that opens two windows 2 and 3 simultaneously, where 2 is modal and 3 is modeless. Figure 2(c) shows a case where the software presents two top-level windows to the user. Window 1 is modal and 2 is modeless. Figure 2(d) shows another case with multiple top-level windows, i.e., 1, 2 and 3. Windows 1 and 2 contain events that open two windows ({4, 5} and {6, 7} respectively) simultaneously.

Figure 3 shows the GUI forest (in this case a single tree) for MS WordPad. Note that the window that is presented to the user when WordPad is launched is called "top-level" and forms the root of the tree. All other windows are either invoked from top-level or from one of the child windows. For example, the window "connect to printer" is invoked from "page setup-2" which in turn is invoked from "page setup-1".

2.3 Flow of Events

The GUI forest in its raw form is not useful for test case generation. We collect additional information during ripping to develop new structures that model the GUI's execution behavior that we call its *flow of events*. Moreover, for testing, we need to develop *units of testing*, i.e., parts of the GUI that can be tested in isolation. The ripping process extracts additional information from the GUI such as event types to develop these structures. We now describe some of this information and structures.

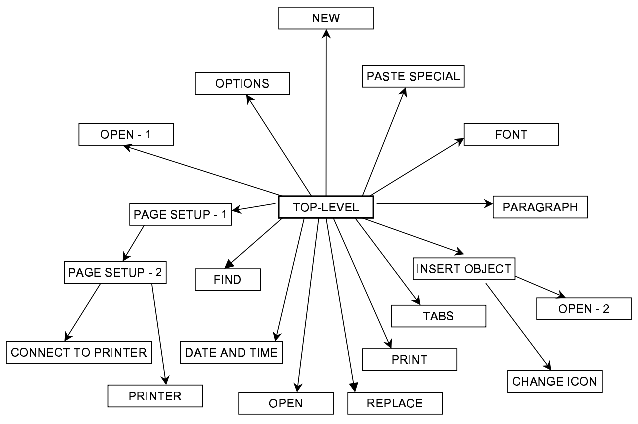


Figure 3. GUI Forest (Tree) for MS WordPad.

To develop units of testing, we exploit the GUI's hierarchy to identify groups of GUI events that can be analyzed in isolation. One hierarchy of the GUI and the one used in this research is obtained by examining the structure of modal windows in the GUI.

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 $\mathcal{RF}$. □

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

Component	Event Types				
	Terminal	Restricted Focus	Unrestricted Focus	System Interaction	Menu Open
Main	1	11	2	69	6
FileNew	2	0	0	2	0
FileOpen	2	0	0	18	0
FilePrint	2	0	0	3	0
FilePage Setup	2	1	0	21	0

Table 1. Some GUI Components of WordPad.

Since components are defined in terms of modal windows, a classification of GUI events is used to identify components. The first class of events, called **restricted-focus events** open *modal windows*. For example, `Set Language` in MS Word is a restricted-focus event. The second class, called **unrestricted-focus events** open *modeless windows*. For example, `Replace` in MS Word 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, called **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 must be explicitly terminated. The most common example of menu-open events are generated by buttons that open pull-down menus. For example, in MS Word, `File` and `SendTo` are menu-open events.

Finally, **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 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.

Event-flow Graphs: A GUI component's flow of events may be represented as a flow graph. Intuitively, an *event-flow graph* represents all possible interactions among the events in a component. An event-flow graph is created by identifying the events in a GUI component. For every event *e*, the events that can be performed immediately after *e* are identified. They are linked with *e* using the **follows** relation.

Definition: An *event-flow graph* for a 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.

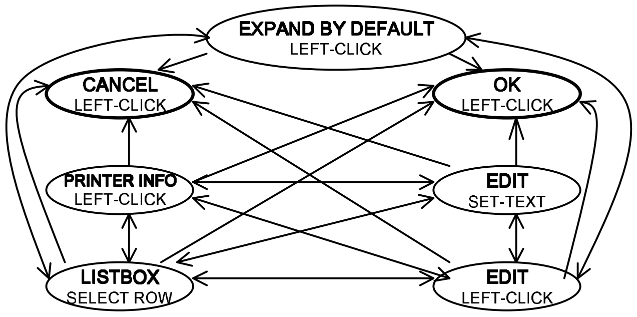


Figure 4. Partial Event-flow Graph for a Component of MS WordPad.

2. **E** $\subseteq$ **V** $\times$ **V** is a set of directed edges between vertices. Event e_j **follows** e_i 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 the "connect to printer" component of MS WordPad is shown in Figure 4. The nodes represent events in the component and the edges show the `follows` relationship.

Integration Tree: Once all the components of the GUI have been represented as event-flow graphs, the remaining step is to identify event flows among components. A structure called an *integration tree* is constructed to identify interactions (invocations) among components.

Definition: Component C_x **invokes** component C_y if C_x contains a restricted-focus event e_x that invokes C_y. □

Intuitively, the integration tree shows the invokes relationship among all the components in a GUI. Formally, an integration tree is defined as:

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 and $\mathcal{R} \in \mathcal{N}$ is a designated component called the `Main` component. $\mathcal{B}$ 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. □

Note that a software's integration tree is very different from its GUI forest; each node in a GUI forest represents a window whereas a node in an integration tree represents a group of windows (called a component, as defined earlier).

Test-case Generation: GUI test cases are sequences of GUI events [12]. Once we have the event-flow graphs and integration tree, we generate test cases by traversing these

```
PROCEDURE DFS-Trees(DFS-Forest $\mathcal{F}$)
  $\mathcal{R}$ /* Set of all root nodes in the forest $\mathcal{F}$ */       1
  FORALL $root \in \mathcal{R}$ DO                                            2
    DFS-Tree-Recursive(root)                                                  3

PROCEDURE DFS-Tree-Recursive(Node n)
  $\mathcal{W}$ = get-child-nodes(n)                                         4
  $\mathcal{W}$ /* Set of child nodes of the node being visited */           5
  FORALL $w \in \mathcal{W}$ DO                                               6
      DFS-Tree-Recursive(w)                                                   7
```

Figure 5. Visiting Each Node of a Forest of Directed Trees

structures and enumerating the events encountered. A large number of test cases can be obtained quickly in this manner. We use *test coverage criteria* to guide the test case generation process. A detailed discussion of test coverage is beyond the scope of this paper. The interested reader is referred to Memon et al. [21] for details.

In the next section, we describe the algorithms of the GUI ripping process.

3 Design of the GUI Ripper

The process of GUI Ripping consists of two steps. First, the GUI of the application is automatically traversed and its structure is extracted by a tool, which we call the **GUI Ripper**. Second, since the implementation may be wrong (after all, that's what is being tested), the extracted information may be incorrect; the tester visually inspects the extracted GUI structure and makes corrections so that the structures confirm to software specifications.

We first describe the algorithm used for the GUI Ripper and then discuss the role of the human tester in inspecting and correcting the extracted structure. We will use a top-down approach to describe our ripping algorithm. Since we use a depth-first traversal (DFS) of the GUI to extract its structure, we will start with a generalized DFS algorithm, tailor an instance of the algorithm for GUIs, and then finally describe specific details of the Windows and Java implementations. We will use stubs in the high-level algorithms that we will later describe in subsequent sections.

GUI Traversal and Extraction Algorithm: As discussed earlier in Section 2, the GUI of an application is structured as a forest. We obtain this structure by performing a **depth-first traversal** of the hierarchical structure of the GUI. We start with a generalized depth-first search algorithm [5] shown in Figure 5 and adapt it for GUIs.

The procedure `DFS-Trees` takes as input a forest, represented as a set of trees. It performs a DFS traversal starting from the root of each tree (lines **2–3**). The procedure `DFS-Tree-Recursive` visits the tree rooted at node *n*. A list $\mathcal{W}$ of all the child nodes of the node *n* is obtained (line **4**). Then a recursive visit for the sub-trees rooted at each of the child nodes is performed (line **6–7**).

We tailor the algorithm of Figure 5 to handle GUI traversal. The resulting algorithm is shown in Figure 6. Two procedures `DFS-GUI` and `DFS-GUI-Recursive` traverse the GUI of the application and extract its structure. The function `access-top-level-windows` (line **1**) returns the list of **top-level windows** in the application under test (AUT). Recall that top-level windows of an application are those windows that become visible when the application is first launched. A GUI tree is constructed for each of the top-level window by invoking the procedure `DFS-GUI-Recursive`. The trees are constructed in the set $\mathcal{GUI}$. At the termination of the algorithm, $\mathcal{GUI}$ contains the GUI forest of the application.

Note that lines **4–7** of Figure 5 has been replaced with lines **5–12** in Figure 6. This is because, for a directed tree, the children of a node can be obtained by invoking the procedure `get-child-nodes`. However, for a GUI application, a node is a GUI window. It may contain several widgets, which in turn, may invoke one or more GUI windows. To obtain a list of all GUI windows that can be invoked from a GUI window *g*, we must query each of *g*'s constituent widgets.

The procedure `DFS-GUI-Recursive` performs a depth-first search of the GUI tree rooted at the GUI window *g*. In line **5** the call to `get-widget-list-and-properties` returns a list $\mathcal{W}$ of the constituent widgets in the GUI window *g*. The function `identify-executable-widgets` in line **6** searches the set $\mathcal{W}$ and returns a list of widgets which invoke other GUI windows. This is because not all of the widgets in $\mathcal{W}$ invokes other GUI windows.

A widget *e* that invokes other GUI windows is executed by `execute-widget` in line **8**. When executed, *e* may invoke one or more GUI windows. The function `get-invoked-gui-windows` in line **9** returns the list of GUI windows invoked by *e*. Note that each of the GUI windows *c* in the set $\mathcal{C}$ are child nodes of the node *g* in the GUI tree. The GUI tree $\mathcal{GUI}$ is updated in line **10**. This is done by inserting each GUI Window *c* from $\mathcal{C}$ as a child node of the GUI window *g*. Lines **11–12** performs a recursive search of the sub-tree rooted at each of the invoked GUI window *c*.

When the procedure `DFS-GUI-Recursive` returns to `DFS-GUI`, the tree rooted at the top-level window *t* is constructed. At the completion of the procedure `DFS-GUI`, the complete GUI forest of the application under test is available in $\mathcal{GUI}$.

The algorithm described in Figure 6 is general and can be applied to any GUI defined in Section 2. In Section 4, we will describe how the high-level functions used in the algo-

```
$\mathcal{GUI}$ /* GUI tree of application under test */
PROCEDURE DFS-GUI(Application A)
    $\mathcal{T}$ = access-top-level-windows($A$)                    1
    $\mathcal{GUI} = \mathcal{T}$                                     2
    /* $\mathcal{T}$ is set of top-level windows in the application */
    FORALL $t \in \mathcal{T}$ DO                                     3
        DFS-GUI-Recursive($t$)                                        4

PROCEDURE DFS-GUI-Recursive(Window g)
    $\mathcal{W}$ = get-widget-list-and-properties($g$)               5
    /* $\mathcal{W}$ is the set of all widgets in the Window */
    $\mathcal{E}$ = identify-executable-widgets($\mathcal{W}$)        6
    /* From $\mathcal{W}$ identify executable widgets */
    FORALL $e \in \mathcal{E}$ DO                                     7
        execute-widget($e$)                                           8
        /* Execute the widget e */
        $\mathcal{C}$ = get-invoked-gui-windows($e$)                  9
        $\mathcal{GUI} = \mathcal{GUI} \cup g$                        10
        FORALL $c \in \mathcal{C}$ DO                                 11
            DFS-GUI-Recursive($c$)                                    12
```

Figure 6. GUI Traversing and Extracting the GUI of an application

rithm may be implemented using Windows and Java API.

Manual Inspection: The automated ripping process is not perfect. Different idiosyncrasies of specific platforms sometimes result in missing windows, widgets, and properties. For example, we cannot distinguish between modal and modeless windows in MS Windows; we cannot extract the structures of the `Print` dialog in Java. Such platform specific differences require human intervention. We provide tools to edit and view the extracted information. We also provide a process called "spy" using which a test designer can manually interact with the AUT, open the window that was missed by the ripper, and add it to the GUI forest at an appropriate location.

Generating the Event-flow Graph and Integration Tree: During the traversal of the GUI, we also determine the event type (discussed in Section 2) by using low-level system calls. Once this information is available, we can create the event-flow graphs and integration tree relatively easily using algorithms described in [12]. We omit details of the algorithms here due to lack of space.

4 Implementation

We now describe the platform-specific details of our two implementations of the GUI Ripper, one for MS Windows GUIs and the other for Java Swing GUIs. We will frequently refer back to the line numbers and high-level functions invoked in the algorithm of Figure 6.

Windows Applications: Before detailing the Windows implementation, we describe some windows-specific details. The Windows Operating System provides a **handle** for all GUI windows and widgets. The handle is an identifier, which uniquely identifies the GUI window or widget. Using the Windows API (*Application Programmers Interface*) it is possible to perform GUI operations such as enumerating the visible GUI windows, enumerating the widgets embedded in a GUI window and detecting the invocation of a new GUI window.

Lines 1–2. The Windows Ripper needs to identify the top-level windows of an application. This is a manual process, where the tester points-and-clicks on the top-level windows. The GUI Ripper, which executes as a background process, records the windows handle of the top-level windows.

Lines 3–4. A Recursive depth-first search is initiated for each top-level window using its window handle.

Line 5. The procedure `get-widget-list-and-properties` returns the list of *all* the widgets in the specified GUI window and their state. It uses the Windows API *EnumChildWindow*, which takes a handle to the GUI window and returns a list of widgets (handles) embedded in it. The handles are then queried for state information of the widgets, such as visibility state, caption, etc.

Line 6. 'Executable' widgets are those that represent restricted-focus events, i.e., those that invoke other GUI windows. The *caption property* of a widget is examined to see if it ends with three dots '...'. For Windows applications, this signifies that the widget is executable.

Lines 7–8. An 'executable' widget is executed by emulating a user's left-click mouse action. The Windows API *SendMessage* is used to send a message to the widget to emulate it.

Line 9. The procedure `get-invoked-gui-windows`, returns the list of GUI windows that are actually invoked by an executable widget. This is implemented using a Windows *hook*. A hook is a mechanism by which a predefined user level functions is called by Windows, whenever a specified GUI event occurs. In our case, this event is the invoking of one or more GUI windows. If the widget invokes GUI windows, $\mathcal{C}$, the handles of $\mathcal{C}$ are sent by Windows to the hook procedure. This handle is then used to analyze the new window.

Line 10. GUI windows that appear in response to executing a widget are child windows of the window containing the widget. The GUI tree being traversed is updated with this structural information.

Lines 11–12. The windows opened by the widget are traversed. Each window is analyzed by the DFS-GUI-Recursive using its unique Windows handle.

The Windows implementation of the GUI Ripper may miss some widgets during the process of ripping. This happens when a widget does not have a Windows handle. Widgets created by the application that bypass the Windows drawing functions usually do not have handles and are missed by the GUI Ripper. After Ripping is complete, the tester may manually add the missed widgets using our spy process.

Java Applications: Java applications do not have a handle and hence cannot be ripped using the Windows Ripper. The Java implementation (**Java Ripper**) is used to rip the GUI structure of applications developed using Java. In applications developed using Java, GUI windows and widgets are instances of Java classes. They are analyzed using Java APIs.

Lines 1–2. From the executable class file(s) of the AUT, the GUI Ripper locates the file containing the `main` class. Using this class, it launches the AUT as an object. The Java API *java.awt.Frame.getFrames()* is used to identify all visible GUI windows (ripper's and those belonging to the AUT). The ripper ignores the windows belonging to itself. The remaining windows are the top-level windows of the AUT.

Lines 3–4. A recursive search is initiated for each top-level window of the AUT using two threads. These are the `Controller` and `Spy` threads. The `Spy` thread analyzes individual GUI windows and their widgets. The `Controller` thread monitors the ripping process and identifies the window to be analyzed by the `Spy` thread.

Line 5–6. The `Spy` thread analyzes each window of the AUT and at the end of the analysis disposes the window. The analysis of the window involves extracting its constituent widgets and their properties. For this we used methods *getComponents()* of class *Container* and *java.awt.Frames.getJMenuBar()* of class *MenuBar*. These methods are then used recursively to get all the widgets (buttons, menu items) that belong to the window. From this array a set of clickable/executable widgets are identified. This is achieved by selecting the widgets that belongs to the *AbstractButton* class family.

Lines 7–8. For analyzing all the windows that belong to the AUT they need to be invoked. A `click` event is executed on the executable widgets. This is done by triggering the `click` event using the Java API *doClick()* of class *AbstractButton*. For example, clicking the menu item `New` on an application will launch *New* window.

Lines 9–10. The new windows that are visible as a result of event are detected using Java API method *java.awt.Frame.getFrames()*. This method returns an array of windows that are tracked by the `Controller` thread. The GUI tree being traversed is updated with this information.

Lines 11–12. With the help of the `Controller` and `Spy` threads the analysis is recursively performed till all the windows of the AUT are analyzed.

Application	Rip Time (Sec)	Ripped Windows	Missed Windows	Manual Effort (mins)	Size (KB)
TerpCalc	29	4	0	5	15.1
TerpPaint	42	7	3	7	24.5
TerpWord	40	10	2	6	53.8
TerpSpreadSheet	89	7	1	7	72.8
WordPad	5	22	2	8	148
Notepad	6	14	2	7	90
Yahoo Messenger	6	18	4	10	159

Table 2. Time for Ripping the GUI of Windows and Java applications

Once all the windows of the AUT are analyzed the Java Ripper generates the GUI forest.

5 Empirical Evaluation

We now empirically demonstrate that the ripping process is *efficient* in that it is fast and requires very little resources and manual effort, and *effective* in that it produces GUI structures that are complete and very close to correct. We also show ripping as part of the overall GUI testing process and compare the time it takes relative to other phases of GUI testing. We have used the extracted GUI structures for automatically generating GUI testing information as part of our GUITAR and DART systems [15].

Ripped Structures: We evaluated the performance of the GUI Rippers on several MS Windows and Java Swing applications. We used Microsoft WordPad, Yahoo Messenger, and NotePad on the MS Windows platform. Our Java test suite is part of an open-source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. It is called TerpOffice[2] and consists of six applications out of which we use four – TerpWord, TerpCalc, TerpPaint and TerpSpreadSheet.

The first step of the ripping process, i.e., extracting the GUI model from the GUI application, is fully automated. It does not requiring any human interaction. Some GUI windows may be missed by the ripper. The tester identifies these windows to the ripper, which can now automatically rip these missed windows.

Table 2 shows the results of ripping the applications. We note that the time taken to rip Java applications is significantly more than Windows applications, although the total time in almost all cases is less than a minute. The time taken to rip an application is directly proportional to the number of windows it contains. This is because, opening windows is a slow windowing process. For example, clicking the `File` → `Open` causes a delay while the application launches the *FileOpen* dialog.

[2]http://www.cs.umd.edu/users/atif/TerpOffice

Application	Rip Application (sec)	Test Case Generation (sec)	Oracle Generation (sec)	Replay Time (sec)	Faults Detected
TerpPaint	42	39	8975	344	2
TerpWord	40	35	4217	344	47
TerpSpreadSheet	89	38	9647	730	12

Table 3. Time for Testing TerpOffice using GUITAR. 1000 Test Cases.

Note that the ripper was able to detect a large fraction of the total number of windows in all applications. Very few windows were missed that had to be manually added later. This process took several minutes. The size of the resulting structures is shown.

Aid to Testing GUI Applications: The GUI Ripper is never used in isolation. We always use it as an important part of large testing tools. We now describe two such tools – GUITAR and DART. GUITAR [12] is a GUI testing framework that we have developed for automated GUI testing. DART [15] is a software that we use for repeated nightly testing of software that have a GUI.

In both GUITAR and DART, a GUI tester extracts the GUI structure of an application using the GUI Ripper, automatically generates test cases for the application based on the extracted information, creates expected output (oracle) for the test cases, executes the test case on the application and determines if the tests ran successfully. Table 3 shows the time taken to perform the entire process of ripping, generating 1000 test cases, generating oracle information and replaying the test cases for three TerpOffice applications for one of our testing experiments. As can be seen from the table, the ripping time is almost insignificant compared to the total time required for testing. As a side-note, the table also shows that we were able to successfully detect faults in the three software.

We note that the GUI Ripper is our most valuable tool in our software testing toolbox. If we did not have the GUI Ripper, we would have spent significant effort in creating the GUI model manually.

6 Related Work

Moore [22] describes experiences with manual reverse engineering of legacy applications to build a model of the user interface functionality. A technique to partially automate this process is also outlined. The results show that a language-independent set of rules can be used to detect user interface components from legacy code. Developing such rules is a nontrivial task, especially for the type of information that we need for software testing.

Systa has used reverse engineering to study and analyze the run-time behavior of Java software [26]. Event trace information is generated as a result of running the target software under a debugger. The event trace, represented as scenario diagrams, is given as an input to a prototype tool SCED [11] that outputs state diagrams. The state diagrams can be used to examine the overall behavior of a desired class, object, or method.

Several different types of representations have been used to generate test information. Anderson and Fickas have used preconditions/postconditions to represent software requirements and specifications [1, 7]. These representations have been successfully used to generate test cases [24, 20]. Scheetz at al. have used a class diagram representation of the system's architecture to generate test cases using an AI planning system [25].

There are various techniques used for testing GUIs [9, 12]. One of our earlier techniques makes use of specifications to generate test cases. In the PATHS [19, 16, 18] system we used an AI planner to generate test cases from GUI specifications. PATHS system uses a semi-automatic approach requiring substantial test designer participation. Our GUI ripping technique is different in that we focus on generating the specifications automatically thereby minimizing test designers involvement.

Chen et al. [4] develop a specification-based technique to test GUIs. Users graphically manipulate test specifications represented by finite state machines (FSM). They provide a visual environment for manipulating these FSMs.

We have successfully used the GUI Ripper software in large GUI testing studies of our DART system [15]. The GUI Ripper was used to generate the GUI structure for several applications. Test cases and *test oracle information* (expected output) [17] were automatically generated from the extracted information.

7 Conclusions and Future Work

Automated testing of software that have a graphical user interface (GUI) has become extremely important as GUIs become increasingly complex and popular. A key step to automatically test GUI software is test case generation from a model of the software. Our experience with GUI testing has shown that such models are very expensive to create manually and software specifications are rarely available in a form to derive these models automatically. We presented a new technique, called GUI ripping to obtain models of the GUI's structure and execution behavior automatically. We represented the GUI's structure as a *GUI forest*, and its execution behavior as *event-flow graphs* and an *integration tree*. We described the GUI ripping process, which is applied to the executing software. The process opens all the software's windows automatically and extracts all their widgets, properties, and values. The execution model of the GUI was obtained by using a classification of the GUI's events. Once the extracted information is verified by a test designer, it is used to automatically generate test cases. We empirically showed that our approach requires very little human intervention. We have implemented our algorithms in a tool called a "GUI Ripper" and have made it available as a downloadable tool.

In the future, we will extend our implementation to handle more MS Windows GUIs, Unix, and web applications. We will also use the GUI ripper for performing usability anlysis of GUIs. It will also be extended for measuring specification conformanc of GUIs.

References

[1] J. S. Anderson. *Automating Requirements Engineering Using Artificial Intelligence Techniques*. Ph.D. thesis, Dept. of Computer and Information Science, University of Oregon, Dec. 1993.

[2] I. Bashir and A. L. Goel. *Testing Object-Oriented Software, Life Cycle Solutions*. Springer-Verlag, 1999.

[3] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1999.

[4] J. Chen and S. Subramaniam. A GUI environment to manipulate fsms for testing GUI-based applications in java. In *Proceeding of the 34th Hawaii International Conferences on System Sciences*, Jan 2001.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, chapter 23.3, pages 477–485. Prentice-Hall of India Private Limited, September 2001.

[6] M. B. Dwyer, V. Carr, and L. Hines. Model checking graphical user interfaces using abstractions. In M. Jazayeri and H. Schauer, editors, *ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 244–261. Springer / ACM Press, 1997.

[7] S. Fickas and J. S. Anderson. A proposed perspective shift: Viewing specification design as a planning problem. In D. Partridge, editor, *Artificial Intelligence & Software Engineering*, pages 535–550. Ablex, Norwood, NJ, 1991.

[8] H. Foster, T. Goradia, T. Ostrand, and W. Szermer. A visual test development environment for GUI systems. In *11th International Software Quality Week*. IEEE Press, 26-29 May 1998.

[9] P. Gerrard. Testing GUI applications. In *EuroSTAR*, Nov 1997.

[10] J. H. Hicinbothom and W. W. Zachary. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting*, volume 2 of *SPECIAL SESSIONS: Demonstrations*, page 1042, 1993.

[11] K. Koskimies, T. Mnnist, T. Syst, and J. Tuomi. Automated support for modeling oo software. In *IEEE Software*, pages 87–94, Jan-Feb 1998.

[12] A. M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.D. thesis, Department of Computer Science, University of Pittsburgh, July 2001.

[13] A. M. Memon. GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):90–91, Aug. 2002.

[14] A. M. Memon. Advances in GUI testing. In *Advances in Computers, ed. by Marvin V. Zelkowitz*, volume 57. Academic Press, 2003.

[15] A. M. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. DART: A framework for regression testing nightly/daily builds of GUI applications. In *Proceedings of the International conference on software maintenance 2003*, September 2003.

[16] 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.

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

[18] A. M. Memon, M. E. Pollack, and M. L. Soffa. Plan generation for GUI testing. In *Proceedings of The Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 226–235. AAAI Press, Apr. 2000.

[19] 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.

[20] 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.

[21] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. In *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 256–267, Sept. 2001.

[22] M. M. Moore. Rule-based detection for reverse engineering user interfaces. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 42–8, Monterey, CA, 8–10 Nov. 1996. IEEE.

[23] R. M. Poston. *Automating Specification-Based Software Testing*. IEEE Computer Society, Los Alamitos, 1 edition, 1996.

[24] M. Scheetz, A. V. Mayrhauser, E. Dahlman, and A. E. Howe. Generating goal-oriented test cases.

[25] M. Scheetz, A. V. Mayrhauser, R. France, E. Dahlman, and A. E. Howe. Generating test cases from an oo model with an ai planning system. In *Proceedings in the Twenty-Third Annual International Computer Software and Applications Conference*, March 2000.

[26] T. Systa. Dynamic reverse engineering of java software. Technical report, University of Tampere, Finland, Box 607, 33101 Tampere, Finland, 2001. http://www.fzi.de/Ecoop99-WS-Reengineering/papers/tarjan/ecoop.html.

[27] A. Walworth. Java GUI testing. *Dr. Dobb's Journal of Software Tools*, 22(2):30, 32, 34, Feb. 1997.