Trends in Model-based GUI Testing

Stephan Arlt* Cristiano Bertolini† Simon Pahl* Martin Schäf†

November 11, 2013

Abstract

This chapter gives an overview of the recent advances in GUI testing. Considering the increasing popularity and fast software development cycles (e.g., desktop and mobile applications), GUI testing gains more importance as it allows us to verify the behavior of a system from the user's perspective. Thus, it can quickly uncover relevant bugs, which a user could face.

Traditional *capture-replay* GUI testing approaches do not meet the demands of developers anymore. Therefore, there is an increasing research activity in *model-based* GUI testing, where the user interaction behavior is simulated using a graph-based model. In the following, we outline different graphical notations to describe feasible user interactions, and methods to generate and execute test cases from these models. We discuss the benefits and limitations of the state-of-the-art in GUI testing research and give a brief outlook about new trends and possibilities to improve the GUI testing automation.

Keywords: Software Testing, System Testing, GUI Testing, Black-box Testing, Grey-box Testing

Contents

1 Introduction **2**

2 Modeling Feasible User Interactions **4**
2.1 Event Flow Graph 5
2.2 Event Interaction Graph 6
2.3 Event Semantic Interaction Graph 7
2.4 Discussion 7

3 Model-based GUI Testing with GUITAR **9**
3.1 GUI Ripper 10
3.2 EFG Construction 12
3.3 Test Case Generator 14

*{arlt,pahls}@informatik.uni-freiburg.de, University of Freiburg, Georges-Köhler-Allee 52, 79110 Freiburg, Germany

†{cbertolini,schaef}@iist.unu.edu, United Nations University, IIST, P.O. Box 3058, Macao

3.4 Replayer 15
3.5 Discussion 16

4 Observations of GUI Events 18
4.1 Shared Event 18
4.2 Independent Event 19
4.3 Visible Event Dependencies 21
4.4 Invisible Event Dependencies 22

5 Extended Model-based GUI Testing 23
5.1 Event-Dependency Graph 23
5.2 Test Case Generation 24
5.3 Implementation 26
5.4 Discussion 29

6 Final Remarks 30

1 Introduction

In software testing, an essential part of system testing is to verify a system through its User Interface (UI) [17]. Simulating possible interactions of a user with the system is a very cost effective way to validate the functionality of a system and to reveal bugs that can be triggered by the user.

Over the past years, Graphical User Interfaces [32, 36] (GUIs) have become the predominant way to interact with electronic devices. A GUI is a collection of *widgets* like buttons, text boxes, etc. During the execution of the system, some widgets are visible and accessible by the user, others might be hidden, or only be available under certain preconditions (e.g., by pressing a certain button first). The user interacts with the system by directly manipulating these widgets (e.g., pressing a button, typing characters in a text box, etc.). Each user interaction with a widget triggers an *event*, which is a message, generated by the operating system that contains information about the widget that was used, the type of interaction (e.g., click, double click), and other data relevant for the interaction (e.g., mouse position). Events might also be generated elsewhere in the system, e.g., by the network adapter when data arrives which has to be picked up the system. However, this chapter limits the overview to events that can be directly triggered by the user.

Each of these events is processed by a particular method in the software, the so-called *event handler*. The event handler polls the events it has been assigned to, and calls other parts of the system according to the data contained in the event. Naturally, GUI testing can only execute those parts of the system that are reachable from the event handlers. Throughout this chapter, events are considered as the atomic unit in our GUI testing. However, for some events, like the input to a text box, the event carries more complex information that could affect which parts of the system will be executed when processing the event. This chapter does not discuss ways to generate test input data and their effect in the GUI testing; it focuses in the test case generation (construction

of the event sequences). The event handler is executed each time that a corresponding message is received. The way a user interacts with a GUI can be seen as a sequence of events, or even as a sequence of calls to the corresponding event handlers.

GUI are always changing, e.g., in the recent years many devices changed from text- or command-based user interfaces [15] (UI) to graphical user interfaces. For example, mobile phones, which traditionally are used over a keypad, nowadays are equipped with touch screens, and the user interacts with purely graphical elements by touching the screen, or even with gestures.

When testing software through its user interface, GUIs represent a special challenge. As opposed to, e.g., a keypad-based user interface, where the number of available buttons limits the possible interactions with the system, the number of possible interactions with a GUI is only limited by the widgets used in the software, and thus might become extremely large.

Until now, the most common way to test GUIs is using *capture-replay tools* (see, e.g., [11]), where a real user executes a set of predefined use cases on the running system. While interacting with the system, all actions such as mouse movement, clicks, or gestures, are recorded as a *test case*. These user interactions then can be replayed anytime to test if the reaction of the system still corresponds to the expected outcome. However, capture-replay tools require a large amount of human interaction. They are vulnerable to changes of visual styles of the application, e.g., if buttons are moved to another position on the screen for the convenience of the user experience, test cases might have to be recorded again. Further, only sequences that have previously been recorded can be tested. This is a very time consuming process and puts a strong limitation on the number of test cases that can be performed.

To make GUI testing more feasible in practice, there has been a reasonable amount of research on automating all aspects of GUI testing, including the generation and execution of test cases. The most visible trend in research is model-based GUI testing, where a graph-based model of possible user interactions with the GUI is used to generate test cases automatically.

In model-based GUI testing, a system with a GUI is seen as an instance of event-driven software [39, 13]. The system (which sometimes is called application layer) performs operations, and the GUI layer facilitates with operations are performed [46]. From the GUI testing perspective, any user interaction with the system is mediated by the GUI without revealing any information about the underlying system which can be seen as *black-box testing* [40], as the structure of the system is not taken into account during testing. In general, black-box testing can be applied to all levels of software testing like unit or integration testing [53]. However, this chapter focuses merely on system level testing.

Model-based GUI testing uses a graph-based model of possible user interactions with the GUI, where each node represents one particular event that can be triggered by interaction with a widget in the GUI. Edges between nodes indicate that two events can be executed consecutively. Hence, a GUI test case can be generated from such a model by selecting any path which starts with an event that can be triggered without any precondition. For each test case, an *oracle* is needed, which decides if this sequence is valid or not. Test oracles can be generated automatically (e.g., from requirements) or user defined. This chapter does not discuss the generation of test oracles.

The user needs to provide an appropriate oracle. The most common oracle, which is also assumed throughout this chapter, is a *crash oracle*, which only checks whether the execution of a given test case does not crash [31, 51]. A crash is considered as an exception or abnormal termination, which the system cannot continue its normal execution. More complex oracles can improve the quality of testing. For example, if the user defines properties based on the functional requirements, then GUI testing can be used as a conformance testing, which checks if specific functions of the system meet their requirements.

Obviously, testing all, possibly infinite, sequences of user interactions is not an option [49, 30, 10]. That is, to efficiently test such a system, techniques that identify *relevant* sequences of events are needed that either cover the most relevant user interactions, or those that lead to program errors.

With a *good* model, large parts of a system can be tested by exercising only a tiny portion of the possible user interactions. However, a non-proper model might generate lots of test cases that execute the same fragment of the system over and over again, without ever executing other fragments. Models for testing GUIs are a vital research area [30, 5, 23] which finds increasing applications in industry [10, 27].

In Section 2, three different graph-based models for GUI testing (e.g., [30, 54, 56]) are presented to illustrate how the usage of models in GUI testing has changed in research in the recent years. Section 3 demonstrates the use of these models in a real GUI testing framework using the example of the GUITAR system[1]. It is important to mention that there are other GUI testing tools that provide similar features as GUITAR, and that could be used to explain model-based GUI testing in a similar way. The decision to use GUITAR is solely based on the background of the authors. Section 4 states several properties of GUIs which can be utilized to further improve GUI testing, and Section 5 gives an outlook about ongoing research which utilizes these properties to identify relevant test cases more efficiently. The chapter closes with remarks on the presented techniques, an outlook on recent trends in research, and open problems in model-based GUI testing.

2 Modeling Feasible User Interactions

In recent years there has been a lot of research on suitable models for automated GUI testing. These GUI models are used to describe the user interface in terms of feasible user interactions. Many formal notations (e.g., Petri Nets, Markov Chains, CSP, etc.) can be used to describe GUIs. However, in current research, graph-based models based on finite state machines are the dominant way to describe GUIs.

Graph-based GUI models are used to represent all possible sequences of interaction of a user with the widgets presented on the screen. Even though these models differ in their details, they all share some basic concepts: a node represents a user interaction with a graphical element (in most cases called *event*), and an edge connects two user interactions that can be taken consecutively.

Some graph-based GUI models store additional information, which can also serve as a documentation of the proper usage of the GUI. For example in [26, 42, 41], the

[1] http://guitar.sourceforge.net/

Unified Modeling Language (UML) is used to represent the GUI structure. UML provides a rich set of features to capture information about user interactions with the GUI and also the data associated with the widgets. However, this data is not used during the test case generation, so for automatic testing, it is sufficient to use a simple graph. Further, the generation of UML models is time consuming and requires expert knowledge. Also, some UML models can be extracted from the code of the GUI [44], some manual steps are required as the generated models are not always precise or complete.

Throughout this chapter we focus on models which can be used within the GUITAR system [25]. Other frameworks exist that provide similar functionality and use similar models. To avoid confusion between the different models we restrict our overview to GUITAR models and only mention a few relevant other approaches at this point. Belli [6, 7] uses another tool called Test Suite Designer (TSD) [3] which also uses a graph-based model called event sequence graph [8, 4, 9] (ESG). Belli uses a terminology related to finite-state machines. However, the semantics of ESG and the graphs presented in this article are very similar and we refer to, e.g., [3] for a detailed comparison.

There are also other variations of finite state machines used for GUI testing, like variable finite state machines (VFSMs) [43], which define a FSM and global variables that are monitored in order to generate test cases. In the following, we restrict our overview to graph-based models used in the GUITAR system.

2.1 Event Flow Graph

One of the most influential models in GUI testing research is the event-flow graph [33, 34, 34, 16, 37, 45] (EFG). An event-flow graph $EFG = \langle E, I, \delta \rangle$ is a directed graph. Each node $e \in E$ is an event in the GUI. Each event in $I \subseteq E$ is the set of initial events which can be executed directly after the application starts. An edge $(e, e') \in \delta$ between two events $e, e' \in E$ states that the event e' can be executed immediately after the event e. In particular, if there is no direct edge between two events e, e', the widget that triggers e' is not available after e is executed. This can, for example, be the case if the widget triggering e' is in a different window (e.g., a modal dialog) which only becomes visible by executing another event e''. So, in this case we have $(e, e') \notin \delta$, but $(e, e'') \in \delta$, and $(e'', e') \in \delta$.

A path in the EFG is a sequence of events, which represents a feasible sequence of user interactions with the GUI. In particular, a path that starts in an initial event represents a sequence of user interactions that can be executed immediately after the application is started. Such a path, together with an oracle represents one test case. This chapter considers only crash oracles and thus, it does not mention it explicitly in the rest of the chapter.

Definition 1. *Given an event-flow graph $EFG = \langle E, I, \delta \rangle$. A test case is a sequence of events $tc = e_0, e_1, \ldots, e_n$, such that $e_0 \in I$ and $(e_i, e_{i+1}) \in \delta$ for all $0 \leq i < n$.*

The EFG can be seen as the most basic GUI model. It only captures which widgets are available after an event has been processed. Thus, it allows us to guarantee that the GUI can accept all sequences of events in our test cases. Later on, we illustrate, how an EFG can be constructed automatically from an application using a GUI ripper [35].

2.2 Event Interaction Graph

Similar to the EFG, an Event Interaction Graph [54, 52, 50] (EIG) is a directed graph where the nodes represent events. The EIG is generated from an EFG. The purpose of an EIG is to find a compact representation of the GUI for a more efficient test case generation. In [52], the concept of EIG is motivated by distinguishing three different types of events:

- *Reachability events*: are used to open windows or menus. For example to open a file in the top menu of an application, the user has to click on `File` which renders a sub-menu, where `Open` can be selected. Both `File` and `Open` emit an event. However, only `Open` is relevant for testing.
- *Termination events*: are events that close dialogs such as `Abort`, and `Cancel`. These events need a special treatment, since they should not appear in the middle of a sequence of events.
- *System Interaction events*: are all other events, such as button clicks, text entry, etc. System Interaction events are the actual events that should be tested.

The EIG is created from the EFG by removing all reachability events. Predecessors and successors of these reachability events are connected by direct edges after removing the reachability event. During the test case generation, sequences of events are selected from the EIG instead of the EFG. For the selected sequences, the test case generation asserts, that all events are system interaction events and termination events only occur at the end of a sequence. As the EIG sequences are not executable (because the reachability events are omitted), the test case generation looks up the shortest possible sequence in the EFG that contains all events occurring in the EIG sequence. Omitting the reachability events results in a smaller model and reduces the number of irrelevant test cases.

The EFG model is created considering all possible user inputs. For example, the user can click in a menu and then select copy. In fact the action of clicking the menu does not have any impact into the system state. However, the options in the menu, for example, *Copy*, *Cut* and *Paste* can impact on the system state and reveal a bug. In this way, the EIG is proposed [54, 52, 50]. The basic idea is that drop off events in the EFG will not directly change the system state. The main assumption is that in most GUIs there are several buttons in one window that are used to carry out completely independent tasks, e.g., a toolbox usually offers *Edit*, *Copy*, *Cut* and *Paste*. However, *Edit* is unlikely to have a side effect on all the other buttons. Thus, it might not be efficient to test all combinations of *Edit* plus one other event.

Figure 1 shows a simple GUI with a menu and 3 options: *Cut*, *Copy* and *Paste*; the corresponding EFG and EIG. In the EFG model all GUI elements are represented in the graph, including the *Edit* option that does not change a system's state but provides access to the menu options. In the EIG the *Edit* option is not represented, since the model represents only the events that can modify the system's state.

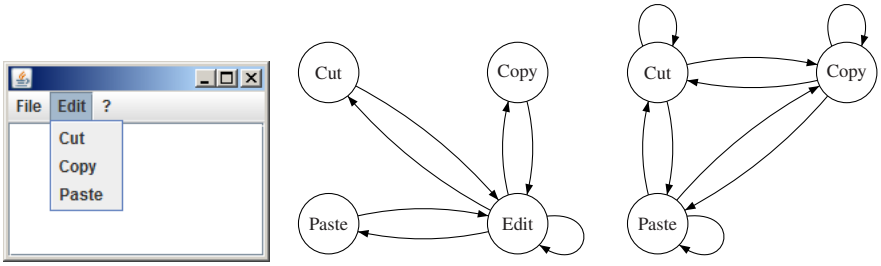

Figure 1: (a) Simple GUI (b) EFG (c) EIG.

2.3 Event Semantic Interaction Graph

While the EFG and the EIG models represent (all) possible sequences of events with a GUI, the Event Semantic Interaction Graph [56] (ESIG) is used to capture relations between widgets that modify each other's properties. Memon *et al.* [56] define a set of *predicates* that represent certain user interactions between widgets over their events. For example, there is an edge between two events e_1 and e_2 in the ESIG, if there is a widget in the GUI whose properties change if the sequence $e_1; e_2$ is executed, but not if only either e_1 or e_2 is executed. In total, there are 6 different predicates used to construct edges in the ESIG. For more details on these predicates, we refer to the related work [56]. Further, the ESIG distinguishes 3 different *context*. That is, for each edge between two events e_1 and e_2, it is recorded, (1) if e_1 and e_2 are in the same window, (2) if e_1 is in a child-window of e_2, or (3) if e_1 and e_2 are in different windows but have a common parent window (in any other case, no edge is created).

The ESIG is created using an EIG. A run-time monitoring checks the current state of GUI widgets after each event. For this state, the changes to the properties of the widgets and the event leading to this state are recorded. For each two consecutive events and their corresponding states, an Event Semantic Interaction (ESI) relationship is defined by evaluating the before mentioned predicates. This results in a matrix that represents the edges of the ESIG.

While the EFG and EIG are used to generate feasible test cases, the ESIG is used to sample the test cases that can be generated from the EFG or EIG. That is, the ESIG aims at identifying an efficient subset of all possible test cases. In that sense the ESIG is a completely different class of model.

2.4 Discussion

Graph-based GUI models serve two main purposes, (1) Enable the automatic generation of executable test cases, and (2) support the test case generation to identify relevant test cases. The EFG is a suitable model for the first purpose. For a given sequence of events, which should be tested, the EFG can be used to validate if all events in that sequence can be executed consecutively, or if some intermediate events are needed. However, the automatic generation of an EFG for a given implementation is a challenge on GUI testing. It requires powerful tools that can traverse the GUI efficiently. Later on, the automatic generation of an EFG in the GUITAR system is explained.

Even though the EFG is suitable to validate the feasibility of a test case, it is not an optimal model for test case generation. The EFG contains many events that should not be subject of a test case. For example, when opening a file in the top menu of an application, usually the user has to click a menu item `File`, which opens a submenu, and then the user has to click `Open`. In the EFG, both, `File` and `Open`, will be represented by a node, but testing `File` in isolation might not be useful. The total number of events in an EFG might be too large to test all sequences of events with a particular length.

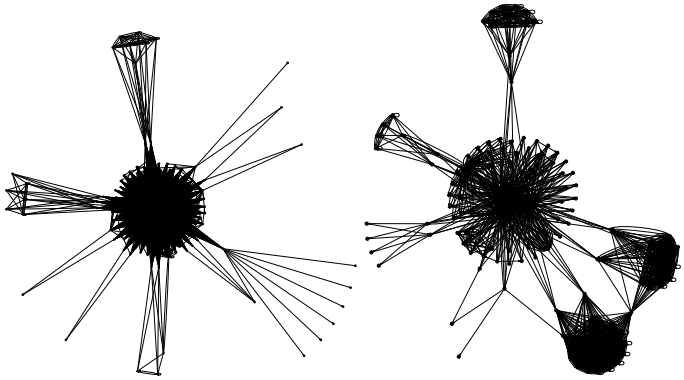

Figure 2: EFG of TerpWord and Rachota.

Figure 2 presents the EFG for the applications TerpWord and Rachota. In TerpWord, the number of generated test cases with a sequence length of 2 is 3,307. In Rachota, the number of generated test cases with a sequence length of 2 is 1,310. Although Rachota only consists of 154 events, the number of sequences with length 5 would be 6,605,912. Assuming that executing one test case takes 10 seconds, all test cases will be finally executed in about 764 days on a single machine. Even for applications of small size, such as TerpWord, the number of test cases generated using only the EFG tends to become prohibitively large [12, 55]. Therefore, it is crucial to find a model that helps to identify and generate relevant sequences of events that can serve as test cases for the user interface of an application.

The EIG confines this problem by using a more abstract model for test case generation. It omits events that should not be tested explicitly, and thus, it results in a more compact model. Reducing the size of the model is a very efficient way to reduce the number of generated test cases. However, abstraction always removes information and thus, it can happen that EIG-generated test cases are not able to cover more code than EFG-generated test cases.

Even though the EIG helps to reduce the number of generated test cases, it does not provide any additional information that can support the test case generation to identifying more *relevant* test cases. Therefore, ESIG extends the EIG by collecting additional data about how widgets influence each other. During the construction of the ESIG, the application is monitored, and if a user interaction with one event influences a property of another widget, this modification is recorded.

The most important difference between the three models is that EFG and EIG are models representing the GUI structure, while the ESIG represents particular user interactions between GUI elements. That is, the first two models are used to identify (all) possible user interactions with the GUI, while the third one is used to sample the set of possible event sequences to identify a special class of test cases. As, even for small programs, the number of test cases, which can be generated from an EIG (or EFG) tends to become large, the combination of models that capture possible user interactions and those that help to identify relevant sequences of user interactions (e.g., ESIG) is extremely important. The importance of these combined models is also visibly stated in recent research, e.g., in [57, 55, 10].

3 Model-based GUI Testing with GUITAR

As mentioned before, the benefit of using a graph-based model of the GUI structure is that the generation and execution of GUI test cases can be automated. In the following, the necessary steps for a fully automated GUI testing are illustrated using the GUITAR system. GUITAR defines both an automated model-based testing process and a corresponding set of tools. In contrast to capture-replay tools, GUITAR automatically builds a model from the GUI, generates test cases from the obtained model, and replays the generated test cases on the GUI. Figure 3 presents an overview of the GUITAR system, which can roughly be subdivided in the following steps:

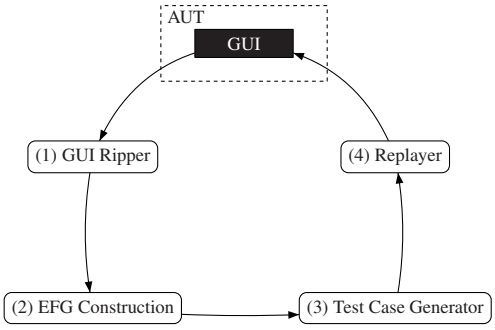

Figure 3: Overview of the GUITAR system.

- **Step 1**: the GUI structure of the application under test (AUT) is recorded using a *GUI ripper*, which represents the widgets of the GUI, such as windows, buttons, and text boxes.
- **Step 2**: the recorded GUI structure serves as an input to the *EFG Construction*, which automatically constructs the EFG that is used for the test case generation later on.
- **Step 3**: the *Test Case Generator* samples paths from the EFG which will serve as test cases in the next step.

- **Step 4**: all generated test cases are executed by the *Replayer*. The *Replayer* creates log files of the executed test cases, which can be investigated, e.g., for potentially thrown exceptions.

Throughout the rest of this section the running example from Figure 4 is used to explain the different steps in GUITAR. The GUI of this application consists of a main window and a dialog. When the application starts, the main window provides the user a button, which opens a dialog. In the dialog, the user can either click the button *Do Something* or *Close Dialog*, which leads back to the main window.

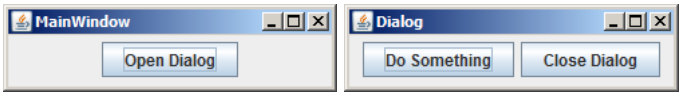

Figure 4: The GUI of the Example Application.

3.1 GUI Ripper

The *GUI ripper* executes the application under test (AUT) and records the GUI structure (see Algorithm 1). A GUI structure represents widgets (e.g., windows, buttons, text boxes) and their corresponding properties (e.g., location, height, width). The recorded widget properties are needed to identify these widgets later during test case execution (see step 4).

Algorithm 1: Ripping of the GUI.

```
   Input: AUT
   Output: GUI structure
 1 begin
 2     run AUT
 3     find main window
 4     recordWindow(main window)
 5     quit AUT
 6     begin recordWindow(window)
 7     store window in GUI structure
 8     foreach widget in window do
 9         store widget in GUI structure
10         trigger widget event
11         if event opens new window then
12             recordWindow(new window)
13         endif
14     endfch
15     end
16 end
```

When executing the AUT, the GUI ripper starts to enumerate all widgets of the main window (line 4) using reflection. For each found widget, the GUI ripper determines the

actual type (that is, Button, Text Box, etc.) and then triggers an event associated with this particular widget (line 10), e.g., a click on a button, or a selection of a menu item. In practice, not every widget is relevant for GUI testing, for example, a `Label` which only displays text, can also receive a mouse click, however, it is not likely that the corresponding event handler is implemented, or will execute real code in the AUT. Therefore, the GUI ripper in GUITAR only supports the following types of events:

- **Action Event:** clicking a button or selecting a check box. Note, the GUI ripper does not simulate a mouse click, but invokes a method of the button object to click the button.
- **Editable Text Event:** inserting a predefined string into an editable text box. The GUI ripper does not perform a key stroke, but invokes a method of the text box object to insert the string.
- **Selection Event:** selection of items in tables, menus, and list boxes, and also nodes in tree views.

If a click on a button opens a new window, for example a dialog, the GUI ripper continues enumerating the widgets of this new window (line 12). So, the ripping process records the GUI structure of the application recursively. It stops, if all found windows have been explored. Note that the ripping process cannot guarantee to find all widgets of the AUT, since the application itself might be hostile or even faulty. For example, if the GUI opens a new window in the background, the GUI ripper will not be able to find it. These problems tend to be of a technical nature and their severity might differ depending on the used platform. The tester can manually improve the recorded GUI structure according to the needed requirements of the AUT.

Beside the properties of the widgets, the GUI ripper records the type of each widget. A widget type states, which changes the corresponding event performs on the GUI. This information is needed, to generate an EFG from the obtained GUI structure (see step 2). During the ripping of the GUI, one of the following types is assigned to each found widget:

- **Terminal:** the event closes the current window.
- **Restricted Focus:** the event opens a modal window.
- **Unrestricted Focus:** the event opens a regular window.
- **System Interaction:** the event does not influence the GUI.

With reflection, functions and attributes of objects can be queried without knowing their actual class. The GUI structure, which is considered as the output of the GUI ripper, consists of one XML document. Figure 5 shows a snippet of the GUI structure of the example application. Each widget has a unique id (*widget id*) to identify it in the further steps in the GUITAR system. Unfortunately, in the actual GUI of an application, widgets do not have this unique *widget id*. Thus, the properties of a widget are used as a heuristic, to identify the widgets during test case execution.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<GUIStructure>
    <GUI>
        <Window>
            <Attributes>
                <Property>
                    <Name>Title</Name>
                    <Value>Main Window</Value>
                </Property>
                <Property>
                    <Name>Modal</Name>
                    <Value>false</Value>
                </Property>
                <Property>
                    <Name>Rootwindow</Name>
                    <Value>true</Value>
                </Property>
            </Attributes>
        </Window>
        <Container>
            <Contents>
                <Container>
                    <Attributes>
                        <Property>
                            <Name>ID</Name>
                            <Value>w3719315156</Value>
                        </Property>
                        <Property>
                            <Name>Class</Name>
                            <Value>javax.swing.JFrame</Value>
                        </Property>
                        <Property>
                            <Name>Type</Name>
                            <Value>SYSTEM INTERACTION</Value>
                        </Property>
                    </Attributes>
                </Container>
             </Contents>
        </Container>
    </GUIStructure>
</GUI>
```

Figure 5: Snippet of the GUI structure in XML.

3.2 EFG Construction

The recorded GUI structure serves as an input to the *EFG Construction*, which automatically constructs the EFG that is used for the test case generation later on. While the GUI structure contains information about windows, widgets, and their corresponding properties, the EFG represents a more abstract view that contains the events and their following events. For the test case generation this abstract view is sufficient. However, for the execution of the test cases, the GUI structure (and the containing information) is needed in order to identify windows and widgets.

The EFG construction iterates over all windows in the GUI structure and creates a single EFG for each window. Later, these EFGs are connected to one EFG representing the entire application.

For each window in the GUI structure, the EFG construction creates an event for the window itself and the containing widgets. Events are identified using a unique *event*

id which is linked to the *widget id* in the GUI structure. Then, the EFG construction connects events of the window based on their widget properties. For instance, if an event e_1 represents a window, and an event e_2 an enabled button in this window, then an edge from e_1 to e_2 is created in the EFG. Assume, that e_2 is associated to a disabled button, then no edge between e_1 and e_2 is created, because the event cannot be triggered if the window appears. Note that the GUI ripper performs a dynamic analysis and thus, the fact if a widget is enabled or disabled during the ripping strongly depends on the environment. For example, user data stored by the application might affect the availability of some widgets, but also the time or execution if a calendar widget is used.

The single EFGs of each window are then connected to one EFG using the widget types described in step 1. For instance, if a widget is type of *Restricted Focus*, then the EFG constructions creates an edge between the event which opens the window, and the event which represents the opened window itself. If a widget is type of *Terminal*, then the EFG construction creates an edge from the event of the opened window itself to the event which formerly opened that window. So, the single EFGs are connected to one EFG of the application.

While the *widget id* is used by the Replayer, the *event id* is considered to distinctly identify an event during test case generation. This enables the support of multiple events per widget in a future work. Figure 6 shows the EFG of the example application.

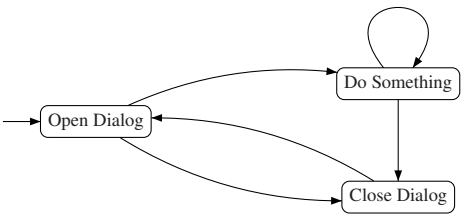

Figure 6: Event-flow Graph of the Example Application.

GUITAR stores the EFG in an adjacency matrix (see Figure 7). The columns and rows of an adjacency matrix contain the events of the application. Each cell expresses, if there exist a relation between the pair of events (value = 1) or not (value = 0). The EFG construction outputs the adjacency matrix to a XML document. A snippet of this XML document is shown in Figure 8.

	Open Dialog	**Do Something**	**Close Dialog**
Open Dialog	0	1	1
Do Something	0	1	0
Close Dialog	1	0	0

Figure 7: EFG Adjacency Matrix of the Example Application.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<EFG>
    <Events>
        <Event>
            <EventId>e3719315156</EventId>
            <WidgetId>w3719315156</WidgetId>
            <Type>SYSTEM INTERACTION</Type>
            <Initial>true</Initial>
            <Action>edu.umd.cs.guitar.event.JFCActionHandler</Action>
        </Event>
        <Event>
            <EventId>e3719315160</EventId>
            <WidgetId>w3719315160</WidgetId>
            <Type>SYSTEM INTERACTION</Type>
            <Initial>true</Initial>
            <Action>edu.umd.cs.guitar.event.JFCActionHandler</Action>
        </Event>
        <Event>
            <EventId>e3719315164</EventId>
            <WidgetId>w3719315164</WidgetId>
            <Type>SYSTEM INTERACTION</Type>
            <Initial>true</Initial>
            <Action>edu.umd.cs.guitar.event.JFCActionHandler</Action>
        </Event>
    </Events>
    <EventGraph>
        <Row>
            <E>1</E>
            <E>1</E>
            <E>1</E>
        </Row>
        <Row>
            <E>1</E>
            <E>1</E>
            <E>1</E>
        </Row>
        <Row>
            <E>1</E>
            <E>1</E>
            <E>1</E>
        </Row>
    </EventGraph>
</EFG>
```

Figure 8: Snippet of the EFG in XML.

3.3 Test Case Generator

From the EFG, a test case can be constructed by selecting an arbitrary path from an initial event (that is, an event without predecessor) to any other event. This can be done manually, for instance, a test engineer starts with selecting one of the initial events in the EFG. This event forms the first event in the test case. Then, the tester follows one outgoing edge to select the second event for the test case, etc. However, manually selecting test cases is time consuming and GUITAR supports different automatic test case generation algorithms. GUITAR is a flexible system, which provides interfaces to implement several test case generators. For brevity of exposure, this section describes only one test case generation algorithm [25] which can be applied for each of the GUI models presented in Section 2.

The test case generator picks a fixed number n of events from the provided model.

These events can either be selected randomly (*random strategy*), or, for example, by picking one event and then all events on a path of length $n-1$ starting in this event (*fixed-length strategy*). Note that the sequence of selected events do not necessarily constitute a path starting with an initial event (that is, the events do not represent an executable test case). Therefore, a path in the EFG has to be found, which starts from an initial event and contains all events from the selected sequence in the right ordering. This involves adding additional events, such as, a prefix path from an initial event to the first event in the sequence, as well as intermediate events that connect two consecutive events in the sequence if no direct connection is available in the EFG. In particular, if the sequence is sampled from a more abstract model, such as EIG or ESIG, the events that are hidden by the abstraction have to be added in order to generate an executable test case.

In Figure 9 all generated test cases of length $n = 2$ for the example application are shown. Thus, each test case covers one edge in the EFG. Events in the bold font represent the actual events under test, while events in the regular font state automatically inserted intermediate events, so-called *reaching steps*. Further, each test case is stored in one XML document to be replayed later on (see Figure 10).

$tc_1 = \textbf{Open Dialog} \rightarrow \textbf{Do Something}$
$tc_2 = \textbf{Open Dialog} \rightarrow \textbf{Close Dialog}$
$tc_3 = \texttt{Open Dialog} \rightarrow \textbf{Do Something} \rightarrow \textbf{Close Dialog}$
$tc_4 = \texttt{Open Dialog} \rightarrow \textbf{Do Something} \rightarrow \textbf{Do Something}$
$tc_5 = \texttt{Open Dialog} \rightarrow \textbf{Close Dialog} \rightarrow \textbf{Open Dialog}$

Figure 9: Example of a Test Case Generation of Length 2.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <TestCase>
3      <Step>
4          <EventId>e3719315160</EventId>
5          <ReachingStep>false</ReachingStep>
6      </Step>
7      <Step>
8          <EventId>e3719315160</EventId>
9          <ReachingStep>false</ReachingStep>
10     </Step>
11 </TestCase>
```

Figure 10: Snippet of a Test Case in XML.

3.4 Replayer

All generated test cases are executed by the *Replayer*. The generated test cases, the EFG and the GUI structure serve as input to the Replayer (see Algorithm 2). For each test case, the Replayer starts the AUT (line 3) and reads the steps from the test case (line 4). A step consists of one event to perform (represented by the event id) and a tag stating if the event is a reaching step or not (true or false). In order to find the event to

perform, the Replayer looks up the event in the EFG using the *event id* (line 6). Once the event id is found, Replayer looks up the corresponding widget of the event in the GUI structure using the *widget id* (line 7). Based on the widget properties, the Replayer tries to find the widget in the GUI (line 8) and to trigger the event (line 10). Similar to the identification of widgets and events in the GUI ripper, the events are triggered using reflection. After the event is triggered, the current state of the application is reported to the oracle (line 12). If the oracle notices an exception, or the widget is not found in the GUI, the test case is marked as *fail* and the Replayer cancels the current test case execution (line 14-15). If all events in one test case have been executed, the Replayer forces the application to terminate and restarts it for the execution of the next test case. Note that each test case is marked as *success* in the beginning. Once all steps of a test case are executed, the Replayer quits the AUT (line 18).

Algorithm 2: Replaying of Test Cases.

```
   Input: Test Cases, EFG, GUI structure
   Output: Result of Test Cases
 1 begin
 2     foreach Test Case in Test Cases do
 3         run AUT
 4         foreach Step in Test Case do
 5             get event from step
 6             look up event in EFG
 7             look up widget in GUI structure
 8             find widget in GUI
 9             if widget is found then
10                 trigger event on widget
11             endif
12             report state to oracle
13             if oracle notices exception or widget is not found then
14                 mark test case as fail
15                 cancel test case execution
16             endif
17         endfch
18         quit AUT
19     endfch
20 end
```

In case the GUI is changing during the development progress, the GUI structure and the EFG might not fit to the AUT anymore. Here, the previous steps (from 1 to 4) have to be executed again, in order to ensure that the Replayer can find the widgets in the AUT. Automatically adjusting the model and the test cases to these changes is still part of ongoing research (e.g., see [22]).

3.5 Discussion

The implementation of GUI testing techniques deals with a lot of technical issues, from the reliable identification of widgets, via the construction of a consistent model, to the

generation of executable test cases from the model. In this discussion, a few technical issues are summarized.

User Settings

As already outlined in the GUITAR system, today's applications maintain user-defined settings, e.g., the recent files opened, or the position and the size of the windows. Those settings are stored in a file when the applications exits, and loaded and applied when the application starts up. User-defined settings are challenging, since they tend to modify the GUI structure, which makes it hard to reliably identify widgets during replay. For instance, the GUI ripper is supposed to record a text editor. After starting the application, the caption of the main window is *Text Editor Application* which is stored as a property in the GUI structure. Next, the GUI ripper opens an existing file, which changes the caption of the main window to *Text Editor - File.txt*. Then, the GUI ripper closes the application, since all windows have been explored. Assume that the text editor stores the name of the recently opened file as a user-defined setting. If the application is restarted by the Replayer, the text editor automatically opens the recent file, which directly sets the caption of the main window to *Text Editor - File.txt*. Since the property of the main window's caption is *Text Editor Application* in the GUI structure, the Replayer will not be able to identify the main windows. Thus, it is challenging to ensure the same preconditions, both for the GUI ripper and the Replayer.

Language Localization

A large number of applications are offered in different languages. This feature is called *Language Localization* and provides the user an automatic or manual selection of the application's input and output language, e.g., English or Arabic. For instance, language localization defines the texts for the application's windows and widgets, as well as their orientation, e.g., from right-to-left instead of left-to-right. For GUI testing, language localization is challenging in two ways:

- A tester must ensure, that the same language settings are defined for the GUI ripper and the Replayer. Usually, the application automatically uses the language of the operating systems. However, an application may allow changing the language during runtime, which should be considered when recording the GUI structure, as this might be triggered by one of the automatically generated test cases, which in turn will cause all succeeding test cases to fail.

- A tester must ensure, that all supported languages of the application are tested. For instance, long captions in widgets may lead to a rearranged GUI structure or even to crashes, if the memory of the caption overflows.

Interleaving

Ensuring the same preconditions for GUI ripping and replaying is crucial. Unfortunately, GUI testing is exposed to non-deterministic behavior, e.g., of the testing environment, which makes it hard to set fix preconditions. For instance, the operating

system notifies the user about new software updates ready to download. This notification might interleave with the GUI of the application that is currently under test. Then, the Replayer is not be able, to identify widgets to perform certain events of the test case. This leads to a failing test case, although the AUT is working properly.

4 Observations of GUI Events

The graph-based models presented so far focus on representing all possible sequences of events with a GUI to allow automatic generation of possible GUI test cases. However, not every sequence might be useful and thus, the presented test case generation algorithms can produce test cases that neither help to increase the code coverage nor will reveal any previously undetected bugs.

This section presents four observations made while applying the presented techniques to the applications TerpWord[2] and Rachota[3]. TerpWord is described in several studies [38, 1, 29]. This application is part of an open-source office suite (TerpOffice) developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. TerpWord is a simple word processor written in Java. Its GUI is implemented using the Java Swing toolkit[4]. Rachota is a time recording system. A user can generate a list of different tasks that should be considered for time recording. If a user starts working on a certain task, Rachota tracks the start time and displays the duration of this task. In the end, the user can check how many hours were spent on each task. Both applications are implemented using the Swing toolkit.

While testing these applications, it turned out that the ability of a generated test case case to cover previously uncovered parts of the program is strongly related to the dependency of the event handlers of consecutive events in that test case. In the following, events are classified depending on their ability to cover new code and to reveal bugs.

4.1 Shared Event

In desktop applications it is a common practice to give the user several different ways to execute one operation. For example, in TerpWord, a user can select the event *Copy* from the menu bar, from the tool bar, or by pressing a key combination. Even though the elements in the user interface are all different, they execute the same code in the application.

Having one event handler that processes events from different widgets is an efficient way to avoid duplicated functionality in the source code. If at least two events share the same handler, it is called a *shared event*.

An example of a shared event handler is shown in Figure 11, which shows a simplified snippet of the *TerpWord* source code. The class constructor creates a menu item `m` (line 4) and a button `b` (line 8). Both created objects obtain exactly the same action

[2] http://www.cs.umd.edu/ atif/TerpOfficeWeb/TerpOfficeV4.0/TerpWord/index.html
[3] http://rachota.sourceforge.net
[4] http://java.sun.com/javase/technologies/desktop/

command, which is done via the method call of `setActionCommand` (line 5, 9), and the same action listener (line 6, 10) using `setActionListener`. An action command represents an unique identifier for a certain event, an action listener is responsible for handling the event when it occurs. Since TerpWord uses the Java Swing toolkit, all events assigned to an action listener are handled in the `actionPerformed` method (line 15). The action command is extracted from an action event object (line 17) and then evaluated for executing the corresponding event handler (line 18, 29).

```
1  public class EkitCore extends JPanel implements ActionListener {
2
3    public EkitCore(){
4      JMenuItem m = new JMenuItem("Copy");
5      m.setActionCommand("textcopy");
6      m.addActionListener(this);
7
8      JButton b = new JButton("Copy");
9      b.setActionCommand("textcopy");
10     b.addActionListener(this);
11
12     // add menu item and button to panel
13   }
14
15   public void actionPerformed(ActionEvent ae){
16     // handle events
17     String command = ae.getActionCommand();
18   }
19 }
```

Figure 11: Example of Shared Events.

From a black-box perspective the user interaction with `m` and `b` are different events. That is, generating GUI test cases from a model that contains both events will result in redundant test cases as replacing `m` by `b` and vice versa results in the same test outcome. Removing one of both events from the model might increase the efficiency of the generated test cases.

Even though shared events seem to be a trivial phenomenon, they have a huge impact on the efficiency of model-based GUI testing. For example, in [1] it is reported that in TerpWord, 28% of the total of event handlers are shared event handlers. That is, it is not only almost certain to create many redundant test cases in such a model, it is also likely that many test cases execute the same event handler several times in a row. For an efficient automation of GUI testing it is crucial to avoid this kind of redundancy in the graph-based model.

4.2 Independent Event

In many desktop applications, a top menu presents the user buttons for the most common tasks, such as creating a new file, copying, or printing. These buttons are available at the same time, and thus can be reached in the GUI models with only one step. However, their effects are more or less independent, i.e., testing arbitrary sequences of these events will have the same effect as testing them in isolation.

In TerpWord, for example, a user can copy selected text to the clipboard via the

event sequence $\langle$`Edit`, `Copy`$\rangle$. The user also can print the document via the event sequence $\langle$`File`, `Print`$\rangle$ in the top menu of the application (see Figure 12). Using the test case generator *SequenceLength* presented in section 3.3, an event sequence $\langle$`Edit`, `Copy`, `File`, `Print`$\rangle$ will be generated for a test case length $n = 2$. Note that the events $\{$`File`, `Edit`$\}$ are reachable steps and do not count regarding the test case length.

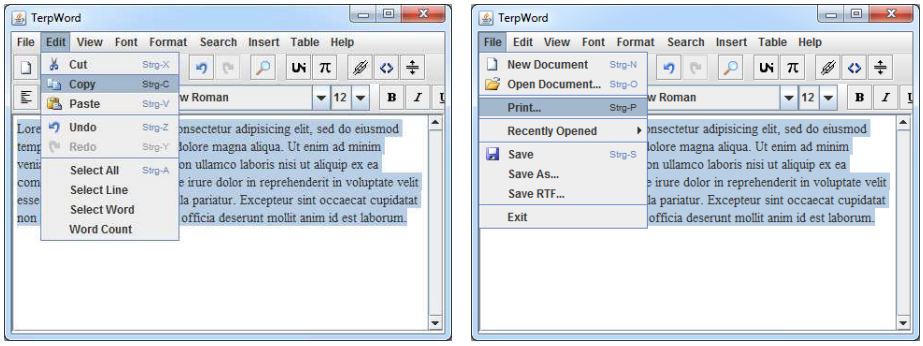

Figure 12: Copying and Printing in TerpWord.

Most often, events such as $\{$`Copy`, `Print`$\}$ do not have a so-called dependency, since `Copy` and `Print` perform completely different tasks. Figure 13 shows a simplified implementation of the events `Copy` and `Print` in TerpWord. In line 11, the selected text is copied to the clipboard by calling the method `copy` of the object `clipboard`. Furthermore, in line 15, the current document is sent to the printer by invoking the method `print` of object `printer`. In this code snippet, the handlers of these two events do not influence each other, and then, do not share a dependency.

```
1  public class EkitCore extends JPanel implements ActionListener {
2
3    public EkitCore(){
4      //...
5    }
6
7    public void actionPerformed(ActionEvent ae){
8      String command = ae.getActionCommand();
9      if ( command.equals("textcopy") ){
10       // copy text to clipboard
11       clipboard.copy(selectedText);
12     }
13     else if ( command.equals("print") ){
14       // print document
15       printer.print(document);
16     }
17   }
18 }
```

Figure 13: Example of Independent Events.

Two events are *independent event* if they do not communicate with each other. That is, they neither exchange data directly (e.g., by passing parameters) nor use any shared

global variables. From a black-box perspective the user interactions `Copy` and `Print` it is obvious, that these two events are independent. For test case generation it would be more efficient to discard those event sequences, which are not dependent. However for an efficient test case generation it is crucial to avoid generation of test cases with sequences of independent events.

4.3 Visible Event Dependencies

Previously, independent events are mentioned, which can be executed in any ordering without affecting each other. For a more efficient test case generation, the exact opposite becomes more interesting. Then dependencies between events are identified, such that preceding one event by another executes different parts of the source code and thus results in improved test coverage.

Events can depend on each other for various reasons. For example, one event might only enabled if a specific event occurred before, or one event depends on the input data of previous event. In this context, event dependency means any kind of user interactions between a set of at least two events.

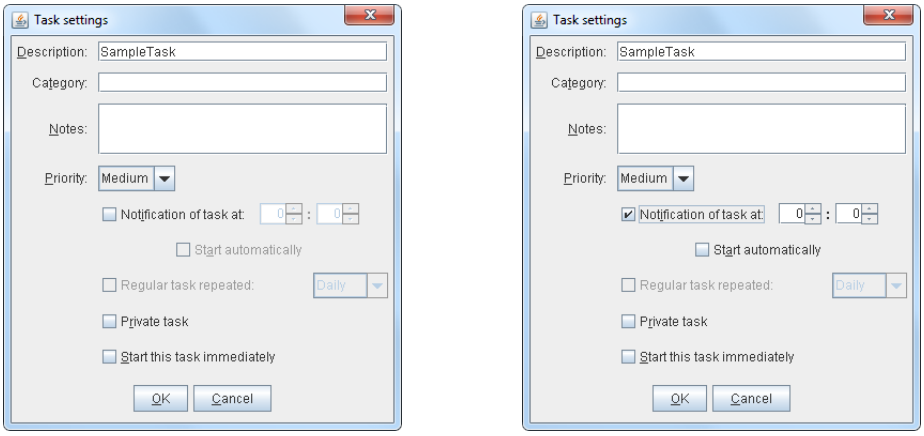

Figure 14: Enabled and Disabled Events in Rachota.

Figure 14 shows the dialog which appears, if the user wants to add a new task. Beside of a couple of text boxes (*Description*, *Category*, *Notes*), the dialog contains a set of check boxes, e.g., *Notification of tasks at:*. When the dialog comes up, this check box is initially deselected. This implies, that some other widgets, i.e., *Start automatically*, are disabled, since it is not allowed to automatically start the notification before no notification time is set in the corresponding text boxes, which are initially disabled as well. The text boxes for setting the notification time, and the check box *start automatically* are enabled, if the user enables the check box *Notification of tasks at:*. So, one event is able to change the properties of a set of other widgets on the GUI, and thus, is able to enable and disable succeeding events. In this example, the check boxes and the text boxes are dependent on a preceding event, because they can only be

used if the notification is enabled. During the construction of the EFG, this behavior of the GUI has to be considered, otherwise some widgets and their corresponding events on the GUI might be discarded for testing.

This dependency between events can be detected without analyzing the source code. Extending the GUI ripper and models to make use of these visible dependencies, e.g., by privileging sequences with visible dependencies during test case generation, can lead to more efficient test cases.

4.4 Invisible Event Dependencies

Events can also influence each other over shared variables. That is, an event handler might execute different operations while handling an event depending on, e.g., the state of a global variable. If another event handler modifies this global variable, then there exists an *invisible event dependency* between the corresponding events.

For example, Figure 15 shows two screen shots of the main window of the TerpWord application. In TerpWord, the user can mainly interact with the application using the menu bar, the tool bar and the main text box. One feature of TerpWord is the split pane, which can be enabled and disabled. Once the split pane is enabled, TerpWord shows the corresponding HTML format of the typed text. When the user calls the operation to copy highlighted text to the clipboard, if the split pane is not yet enabled, TerpWord copies the text of the upper text box into the clipboard, otherwise the text of the lower text box is copied to the clipboard.

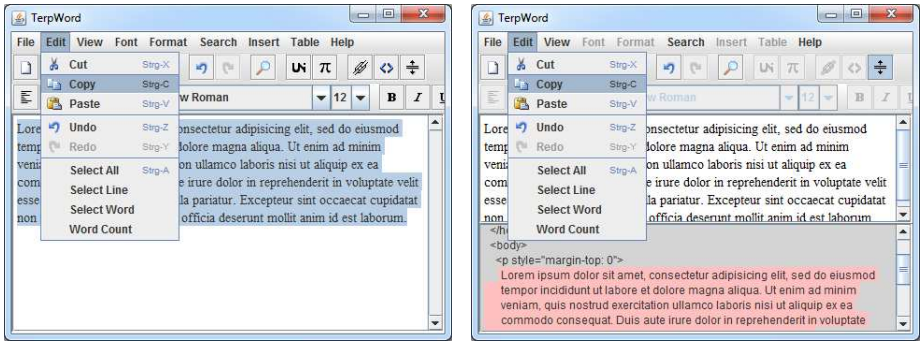

Figure 15: Copying in TerpWord.

This example goes beyond the previously stated example of enabled and disabled events in Rachota. Here, the same event, i.e., copy some text into the clipboard, is triggered, but it leads to different functionality. Thus, the event *Copy* has also a dependency to preceding events triggered on the GUI. From a black-box perspective, it not possible to identify the dependency of the event *Copy* without analyzing the data modified by the application.

5 Extended Model-based GUI Testing

The previous observations can be used to sample the possible sequences of events with a GUI more efficiently. However, the observations relate to properties of the source code, which are not available when testing the system from a black-box perspective. That is, in addition to the previously mentioned models of possible user interactions (EFG, EIG and ESIG), another graph-based model has to be created from the source code of the system. This model helps to consolidate events that execute the same event handlers (shared events), avoid the generation of sequences of independent events, and capture data dependencies between events (visible/invisible event dependencies). Even though this model is generated from the source code, the test case generation itself will only consider the resulting model source code of the AUT. That is, the resulting testing approach can be seen as a grey-box GUI testing approach.

5.1 Event-Dependency Graph

The grey-box approach utilizes two different models of the GUI during test case generation. It uses an *event-flow graph* (EFG) [30] to represent the possible sequences in which events can occur, and an *event-dependency graph* (EDG) to represent data dependencies between events (e.g., an event writes a field which is later read by another event). Both models are directed graphs where each node corresponds to one event in the GUI. The key idea of the EDG model is to find out dependencies between events.

An event-dependency graph $EDG = \langle E, \psi \rangle$ is a directed graph where, like in the EFG, each node in E represents a GUI event. Note that in contrast to the EFG, an EDG does not have initial events, since it represents data dependency and not control-flow. An edge $(e, w, e') \in \psi$ is labeled with a weight w. The weight $w \in \mathbb{N}^+$ indicates the data dependency between e and e'.

The edge value is computed as follows: All fields which are written in the event handler of e are collected in a set W. All fields that are read in the event handler of e' are collected in a set R. For each event handler, it follows potential method calls, collects these fields, and places them in set W and R respectively. The edge from e to e' is labeled with the size of the intersection of these set $|R \cap W|$.

A path $\pi = e_i \dots e_j$ in the EDG represents a sequence of events, where the execution of one event always changes fields which are read by the succeeding event. However, it is not necessary that two events in question can be executed consecutively in the GUI. The benefit of these sequences is that the execution of one event might change relevant fields for the execution of its successor and causes this one to execute other code fragments. This can lead to a higher code coverage and further reduce the amount of code that is tested redundantly.

Since the EDG has no initial event, and succeeding events on a path in the EDG might not be directly executable in the GUI, one EDG path is an *abstract test case* defined as:

Definition 2. *Given an event-dependency graph $EDG = \langle E, \psi \rangle$. An* abstract test case *is sequence of events $\pi = e_i, \dots, e_j$, such that $(e_k, e_{k+1}) \in \delta$ for all $i \leq k < j$.*

Algorithm 3: Construction of the EDG.

Input: P : Program,
$\langle E, I, \delta \rangle$: Event-flow graph
Output: $\langle E', \psi \rangle$: Event-dependency graph

```
1  begin
2      E' = E
3      W = {},R = {}
4      foreach (e in E) do
5          W = getFieldsWritten(e, P)
6          foreach (e' in E) do
7              R = getFieldsRead(e', P)
8              if ((R ∩ W) ≠ ∅) then
9                  w = |R ∩ W|
10                 ψ = ψ ∪ (e, w, e')
11             endif
12         endfch
13     endfch
14 end
```

Algorithm 3 shows how the EDG is constructed. The algorithm takes the program P and the corresponding event-flow graph EFG as input, and returns an event-dependency graph EDG. Since both EFG and EDG refer to the same set of events, E is copied to E' (line 2). Then, it iterates over all pairs of events e, e' (line 4).

The method `getFieldsWritten` is called and returns a set W of all fields that are written during the execution of the event handler of e (line 5). Then, the method `getFieldsRead` is called that returns a set R of all fields which are read during the execution of the event handler of e' (line 7).

If the intersection of R and W is not empty (line 8), a new edge is added to the EDG that is labeled with the size of the intersection (line 10). Note that the algorithm does not create an edge between events if the intersection of R and W is empty. In this case, there is no data dependency between both events and thus, they are not directly connected (otherwise the EDG would be fully connected).

5.2 Test Case Generation

The test case generation is built out of two consecutive steps. First, it selects potentially interesting sequences of events, called *abstract test cases*, from the EDG using Algorithm 4. Second, it uses the *abstract test cases* to generate executable test cases from the EFG using Algorithm 5.

Algorithm 4 takes an EDG and two parameters as input: len gives the maximum length of the *abstract test cases* to be generated, and top gives the maximum number for *abstract test cases* to be generated for each event. The algorithm returns a set Π of *abstract test cases*. These are later used for generating executable test cases.

For each event $e \in E$ a new set Π' of *abstract test cases* is created, which initially is empty (line 2). As long as the size of this set is smaller than top (line 5), further *abstract*

Algorithm 4: Generating *abstract test cases*.

Input: $\langle E, \psi \rangle$: Event-dependency graph,
len : max length of *abstract test case*,
top : max number of *abstract test cases* per event
Output: Π : set of *abstract test cases*

```
1  begin
2      Sequences of events Π = {}
3      foreach Event e ∈ E do
4          Sequences of events Π' = {}
5          while |Π'| < top do
6              Sequence of events π = e
7              Event e' = e
8              while |π| < len ∧ post(e') ≠ {} do
9                  e' = bestSucc(e', Π)
10                 π = π • e'
11             endw
12             if π ∈ Π then break
13             Π' = Π' ∪ {π}
14         endw
15         Π = Π ∪ Π'
16     endfch
17     return Π
18 end
```

test cases are added (line 6). Each such *abstract test case* π initially contains only e (as it looks for sequences of events that start on e). While the length of π is smaller than len (line 8), and the first element of π still has successors, the method `bestSucc` is used to find the best possible successor and add it to the beginning of π (line 9). The method `bestSucc` uses a *minimax* strategy to identify the best successor, unless this successor will lead us on a path which is already in Π'. In this case, `bestSucc` returns the second best choice. The minimax strategy is used to minimize the selection of events with low dependencies.

The loop in line 5 terminates either, if it has collected top *abstract test cases* that start with the event e or, if the algorithm detects a path twice (line 12). In that case, `bestSucc` cannot find a suitable path that has not been visited so far.

For each *abstract test case* in Π, at least one executable test case is generated. However, the *abstract test cases* are not necessarily executable, as consecutive events might have no direct connection in the EFG. Therefore, Algorithm 5 finds one EFG path for each of these *abstract test cases*, which starts in an initial event of the EFG. Note that the only case, where such a path does not exist is, if the application is terminated between two events execution. In that case, the sequence is split into two sequences that later on are tested immediately after each other.

Algorithm 5 takes an EFG and the set Π of *abstract test cases* computed by Algorithm 4 as input, and returns a set of paths in the EFG which start in an initial event. For each sequence of events $e_i \dots e_j$ in Π (line 3), a path tc is created (line 5). The

Algorithm 5: Conversion from *abstract test cases* to *executable test cases*.

Input: $\langle E, I, \delta \rangle$: Event-flow graph,
Π : set of *abstract test cases*
Output: T : Set of *executable test cases*

```
1  begin
2      Sequences of events T = {}
3      foreach Sequence e_i, ..., e_j in Π do
4          pick e_0 from I
5          Path tc = shortestPath(e_0, e_i)
6          for k = i to j − 1 do
7              tc = tc • shortestPath(e_k, e_{k+1})
8          endfor
9          T = T ∪ {tc}
10     endfch
11     return T
12 end
```

shortest path is picked from an event $e_0 \in I$ to e_i, and then iterate over the events in the *abstract test cases* and always add the shortest path between succeeding events to tc (line 7). Then tc is added to the set T (line 9). Since paths in T start in an initial event of the EFG, it can immediately executed as a test case.

5.3 Implementation

GUITAR was extended to support grey-box GUI testing. A bytecode analysis that collects additional information about the data dependency between event handlers is added. The new graph-based model (EDG) for test case generation that utilizes this information is generated, and an extended test case generation is implemented. Figure 16 presents an overview of the GUITAR system including the grey-box extensions. The grey-highlighted steps in the overview represent the parts of the GUITAR system that are modified or added.

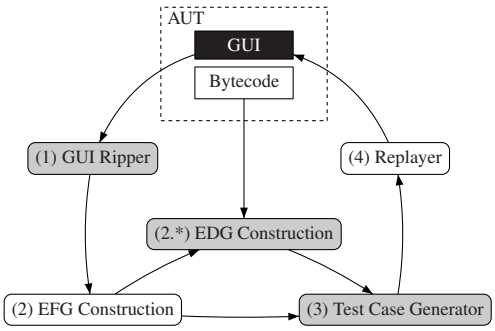

Figure 16: The GUITAR System, including the Grey-box Extensions.

The GUI ripper is enhanced, such that, for each widget, the event handlers associ-

ated with this widget are stored in the GUI structure. This information is needed during the bytecode analysis, which is performed as part of the EDG construction. The bytecode analysis records all program fields that are read or written by each event handler, that is, the functions `getFieldsRead` and `getFieldsWritten` in Algorithm 3. These data dependencies between the event handlers are used to construct the EDG. Then, the test case generation utilizes the EFG obtained during the GUI ripping step and the EDG to construct executable test cases as described in Algorithm 4.

As the main steps of the extended GUITAR system only contain minor changes compared to the original system described in Section 2, the rest of this section focuses on the bytecode analysis which represents the most relevant extension.

Bytecode is a platform-independent binary format and exists for many programming language and runtime systems. Bytecode can be considered as the target format of compiled Java Programs [28]. During the execution of Java programs, the Java Virtual Machine maps platform-independent instructions of the bytecode into concrete machine instructions of the current system. Bytecode instructions are oriented towards the high-level programming language, from which they are compiled. Often this makes the bytecode so similar to the program it originated from, that it might be possible to reconstruct the complete source code from the bytecode.

Since bytecode is the executable format of Java programs, it provides both the opportunity to run and to analyze those programs out of the box. A clear benefit of bytecode for static program analysis is that a certain degree of validity of the bytecode can be assumed, and many errors, such as syntax or type errors, do not have to be considered during the analysis.

Figure 17 shows a Java class and the corresponding translation into bytecode. The Java class consists of an instance field (or field) x, a static field y, and 4 methods ($getX$, $setX$, $increaseY$, and $EventHandler$). Bytecode has the same flat structure like assembly instructions, which can easily be traversed in a sequential fashion.

In bytecode, each class has a fully-qualified name which allows to distinctly identify them. This is important, for example, if the same class name is used in different packages. That is, from the bytecode it is possible to identify the object-oriented structure of the original Java program. Thus, information about classes, methods and fields are still available in the bytecode.

The Java Virtual Machine is a stack machine, that is, local fields no longer have a name and are represented by slots on a stack. Therefore, all instructions read their operands as an input from the stack and write the results of the execution back on the stack. Some instructions receive additional non-stack operands, which supply references to field names, method signatures or other values from a constant pool.

In the implementation the bytecode analysis framework ASM[5] is used. Other frameworks such as BCEL[6] or Soot[7] could be used equally well. First, the bytecode is translated into a syntax tree. Then, the syntax tree is traversed using the visitor design pattern [18]. Because of the flat structure of bytecode, the instructions of a method are visited sequentially via a loop.

[5] http://asm.ow2.org/
[6] http://commons.apache.org/bcel/
[7] http://www.sable.mcgill.ca/soot/

Java:

```
1  public class Example{
2
3    int x;
4
5    static int y;
6
7    public int getX() {
8      return x;
9    }
10
11
12   public void setX(int x) {
13     this.x = x;
14   }
15
16
17
18   public int increaseY() {
19     return y++;
20   }
21
22
23
24
25
26   public void EventHandler(){
27     setX(1);
28   }
29
30
31 }
```

Bytecode:

```
1  public class Example{
2
3    i x
4
5    static i y
6
7    public getX()i
8      aload 0
9      getfield Example.x : i
10     ireturn
11
12   public setX(i)v
13     aload 0
14     iload 1
15     putfield Example.x : i
16     return
17
18   public increaseY()i
19     getstatic Example.y : i
20     dup
21     iconst_1
22     iadd
23     putstatic Example.y : i
24     ireturn
25
26   public EventHandler()v
27     aload 0
28     iconst_1
29     invokevirtual Example.setX(i)v
30
31 }
```

Figure 17: Comparison of Java Source Code and Bytecode.

The bytecode analysis investigates all classes of the AUT and stores the information in a class database (*ClassDB*). It is possible to restrict the set of classes that should be considered for the bytecode analysis. Usually, only the application classes are analyzed and third-party libraries are ignored. However, in the implementation a so-called *scope* can be defined. The scope defines the classes (.class files or JAR archives) of an AUT that are analyzed. The ClassDB models the dependencies between fields, methods and classes of the AUT. Further, the ClassDB is used as a data container and as an intermediate step towards the construction of the EDG.

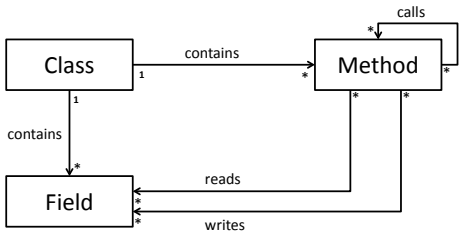

Figure 18: Relationship between fields, methods, and classes.

In the following, an overview of the bytecode analysis is provided, which artifacts

of the bytecode are used, and how they find their way into the EDG construction. Figure 18 illustrates the relationships between classes, methods, and fields (and also represents the ER-diagram of the ClassDB used in the bytecode analysis).

Classes are the starting point for the analysis, because they contain both methods and fields. The relationship *contains* describes, that methods and fields are distinctly associated with their classes.

First, the methods of each class in the AUT are analyzed. Note that it is important to inspect all methods, and not only those which are considered as event handlers. For instance, in method `EventHandler` in 17, the method `setX` is called, which performs write access on the field `x`. In order to recognize this write access, the bytecode analysis has to determine, that method `EventHandler` called method `setX`. Thus, there exist a recursive relationship *calls* between methods.

Then, for each method the bytecode analysis fetches all fields that are read and written in the method and stores them in two distinct lists. These lists also contain the fields of the called methods. Java distinguishes between instance fields and class fields (marked with the keyword *static*). While instance fields belong to objects, class fields are not related to a particular instance of a class. For both types of fields, there exist a read and write instruction in the bytecode (see 17): `getfield` gets the value of an object field; `putfield` set a value of an object field; `getstatic` gets the value of a static field; `putstatic` sets a value of a static field. Both instance fields and class fields are treated the same way. That is, not only class fields are mapped to a certain class in the ClassDB, but also instance fields. Moreover, instance fields are not mapped to their objects. Further, the bytecode analysis does not distinguish between instance and class methods and, thus, is not reliable regarding polymorphism. In order to address these issues, a more in-depth data- and control-flow analysis is subject to future work.

Once all classes, methods and fields are visited and stored in the ClassDB, the Algorithm 3 can use this information to construct the EDG. For instance, if the algorithm requests the `getFieldsRead` and `getFieldsWritten` for a certain event e, the ClassDB aggregates all called method within the event handler of e. For each called method, and for the event handler itself, the read and written fields are collected and returned to the EDG construction. In this way, possible dependencies between events can be obtained. Further, due to this shallow analysis of the bytecode, the computation time for building the ClassDB and the EDG construction is low, even for big applications.

5.4 Discussion

Similar to the ESIG discussed in Section 2.3, the EDG is used to sample a set of possible sequences of events with the GUI given by the EFG. While the ESIG captures dependencies between events based on a set of predefined predicates, the EDG captures simple data dependencies. Since, until now, no comparable benchmarks are available, it is not clear if predicates are too restrictive or data dependencies are too general. Both, ESIG and EDG-based testing can be seen as grey-box testing as both require information about the source code. However, the ESIG is generated using run-time monitors and thus depends on the used GUI ripper, while the EDG is generated using

static analysis. In general, an EDG could also be generated using runtime monitors and vice versa, but at this point there are no such implementations available.

It is important to note that the grey-box approach does not explore the state space of the AUTs [24, 2, 14], it explores the possible user interaction of an AUT. Both approaches (state space and interaction exploration) can be used together. Moreover, the grey-box approach uses sequences of events, which eventually bring the system to a failure state. In this way, the information used in grey-box approach is based on possible user interactions with the GUI instead of information about the system state space as in software model-checkers.

There are clear benefits when more information about the AUT is available and incorporated into the test case generation. It allows the test case generator to generate more efficient test cases and to avoid redundant ones. However, mining information from the source code of an application (that is, static analysis) has several limitations. First of all, the source code has to be available in a form that supports the analysis. Further, the information that can be mined from source code is very limited by the complexity of the problem and always is a trade-off between precision and performance.

The extent to which the source code of an application can be used for test case generation is flexible. For example, symbolic execution for testing GUI based software could be considered [20][19] to find adequate inputs for test cases. While symbolic execution is a powerful technique to find precise input values, its applicability is limited due to the complexity of the used algorithms. Therefore, limiting the analysis of the source code to identify simple data dependencies without tracking the actual value of fields is more applicable for reasonably sized applications. However, the sweet spot between mining detailed information for more efficient test cases and an efficient and scalable implementation remains the subject of research.

The bottleneck of any automated GUI model construction is the GUI-ripper. Computing a model of the GUI structure can be understood as the problem of computing which parts of the source code are reachable. This problem is not decidable for the general case. Thus, any GUI-ripper might introduce some imprecision. That is, why automated GUI testing has to be judged based on its practical use rather than on its theoretical contribution.

6 Final Remarks

Model-based testing of GUIs offers the great opportunity to automate large parts of the GUI testing process. Research has shown that models of possible user interaction with a GUI can be generated automatically, and also test cases can be executed without any manual interaction.

With the increasing automation of GUI testing, new challenges arise. The generated models tend to become extremely large for real software, such that a full exploration of all possible user interactions, even if restricted to a particular length, tends to be impossible. Newer models such as the ESIG tackle this problem by representing only a subset of all possible user interactions, which satisfy some properties that are assumed to be interesting for testing. Other approaches such as the grey-box extension move in

same direction. For the future development of automated GUI testing it is mandatory to find proper models that help to sample the set of possible user interactions with the GUI. These models will need more information about the GUI and the purpose and dependencies between the different elements in the GUI. From the current state-of-the-art, a combination of shallow, and scalable static analysis, together with model-based (black-box) testing seems to be the most promising direction.

Incorporating information from the source code in the GUI testing automation can improve many parts of the GUI testing process. The construction of the model, which right now is done by monitoring the execution during runtime could utilize information about the widgets defined in the source code, or data dependencies between widgets to find a model more efficiently and probably identify parts of the model that require particular user settings.

Future GUI models could use static analysis to identify under which preconditions an event handler executes different source code fragments and reaches higher code coverage. Static analysis could also help to avoid unnecessary test cases that will speed up the entire testing process. Similar approaches already exist in unit testing [21].

During the test case execution, the source code could be analyzed to compute test input data, which steers the application towards a particular state (for example a crash). Generating appropriate input data for test cases is crucial to increase the code coverage and to find bugs efficiently [48]. White-box testing, e.g., *Pex* focuses on the generation of input data for unit tests using dynamic symbolic execution [47]. The technique explores the program's source code and uses a constraint solver to produce input data, which achieves high code coverage. This technique could be adapted for GUI testing, in order to select sequences of dependent events, and to generate useful input data to the test cases.

Unfortunately, static analysis is a complex topic, and mining information from source code can be computational expensive or even impossible. For example, for an application which allows the user to modify data in a complex database, or which uses services from the web, it will be impossible to identify proper input data using static analysis as the amount of data will either be too large, or parts of the source code will not be available. Therefore, finding the right balance between a detailed analysis of the source code and testing will remain an important research topic.

Throughout this chapter, only sequential programs have been considered. For parallel programs there are several problems, which have not been considered here. For example, in an e-mail client, any sequence of user interaction might be interleaved by an incoming e-mail. The models presented in this chapter cannot cover this situation. New models are required which are able to consider events, which are triggered by the environment, and not by the user.

In conclusion, model-based GUI testing has made advances in the past years and is already frequently applied in practice. GUI models are improving in order to create samples from feasible user interaction models to be more efficient and effective. Probably, sampling models with specific goals can achieve better results, e.g., sampling only regions with new requirements or only with a determined dependency between widgets. Moreover, technical issues related to the GUI ripper and Replayer need further research effort to minimize the testing process when GUI evolution is considered. GUIs are evolving and different widgets, platforms and environments are emerging. In

this way, it is still hard to validate and extend current models and techniques due to the lack of available benchmarks. A recent initiative, called COMET - Community Event-based Testing [16], is moving efforts to create a free and open benchmark for GUI testing. However, many pressing problems, such as test input, computation of oracles, and models suitable for parallel systems still require additional research effort.

Acknowledgment

This work is supported by the research projects ARV and EVGUI funded by the Macau Science and Technology Development Fund.

References

[1] Stephan Arlt, Cristiano Bertolini, and Martin Schäf. Behind the Scenes: An Approach to Incorporate Context in GUI Test Case Generation. In *TESTBEDS 2011: Proceedings of the International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software*, Berlin, 2011. IEEE Computer Society.

[2] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. What You See Is Not What You eXecute. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, pages 202–213. 2008.

[3] F. Belli, M. Beyazit, and N. Güler. Event-Based GUI Testing and Reliability Assessment Techniques – An Experimental Insight and Preliminary Results. In *TESTBEDS 2011: Proceedings of the International Workshop on TESTing Techniques & Experimentation Benchmarks for Event-Driven Software*, Berlin, 2011. IEEE Computer Society.

[4] F. Belli, N. Nissanke, Ch. J. Budnik, and A. Mathur. Test Generation Using Event Sequence Graphs. Technical Report 6, University of Paderborn, Angewandte Datentechnik, Aug 2005.

[5] Fevzi Belli. Finite-State Testing and Analysis of Graphical User Interfaces. In *ISSRE*, pages 34–43. IEEE Computer Society, 2001.

[6] Fevzi Belli. Finite-state testing and analysis of graphical user interfaces. In *ISSRE*, pages 34–43, 2001.

[7] Fevzi Belli. Goal-driven, scalable generation of complete interaction sequences for testing graphical user interfaces. In *IEA/AIE*, pages 919–920, 2001.

[8] Fevzi Belli, Christof J. Budnik, and Lee White. Event-based modelling, analysis and testing of user interactions: approach and case study: Research Articles. *Softw. Test. Verif. Reliab.*, 16:3–32, March 2006.

[9] Fevzi Belli, Christof J. Budnik, and Lee White. Event-based modelling, analysis and testing of user interactions: approach and case study: Research articles. *Softw. Test. Verif. Reliab.*, 16:3–32, March 2006.

[10] C. Bertolini, A. Mota, E. Aranha, and C. Ferraz. GUI Testing Techniques Evaluation by Designed Experiments. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 235–244, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[11] Rex Black. *Advanced Software Testing - Vol. 1: Guide to the ISTQB Advanced Certification as an Advanced Test Analyst (Rockynook Computing)*. Rocky Nook, 2008.

[12] Penelope A. Brooks and Atif M. Memon. Automated GUI testing guided by usage profiles. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 333–342, New York, NY, USA, 2007. ACM.

[13] Renee C. Bryce, Sreedevi Sampath, and Atif M. Memon. Developing a Single Model and Test Prioritization Strategies for Event-Driven Software. *IEEE Transactions on Software Engineering*, 37(1):48–64, 2011.

[14] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. In *ASE*, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[15] Jung-Wei Chen and Jiajie Zhang. Comparing text-based and graphic user interfaces for novice and expert users. *AMIA Annu Symp Proc*, pages 125–9, 2007.

[16] COMET. Community Event-based Testing. http://comet.unl.edu/.

[17] Alan J. Dix, Janet E. Finlay, Gregory D. Abowd, and Russell Beale. *Human-Computer Interaction*. Prentice Hall, 2004.

[18] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, 1995.

[19] Svetoslav Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne Perry. Event Listener Analysis and Symbolic Execution for Testing GUI Applications. In *ICFEM*, pages 69–87, 2009.

[20] Svetoslav R. Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne E. Perry. Test Generation for Graphical User Interfaces Based on Symbolic Execution. In *AST*, pages 33–40. ACM, 2008.

[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[22] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving gui-directed test scripts. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 408–418. IEEE, 2009.

[23] A.M.P. Grilo, A.C.R. Paiva, and J.P. Faria. Reverse Engineering of GUI Models for Testing. In *5th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1 – 6. IEEE Computer Society, 2010.

[24] Alex Groce and Willem Visser. Heuristics for Model Checking Java Programs. *STTT*, 6(4):260–276, 2004.

[25] Daniel Hackner and Atif M. Memon. Test case generator for GUITAR. In *ICSE '08: Research Demonstration Track: International Conference on Software Engineering*, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Ying Hou, Rong Chen, and Zhenjun Du. Automated GUI Testing for J2ME Software Based on FSM. In *Proceedings of the 2009 International Conference on Scalable Computing and Communications; Eighth International Conference on Embedded Computing*, SCALCOM-EMBEDDEDCOM '09, pages 341–346, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Antti Jääskeläinen, Tommi Takala, and Mika Katara. Model-based GUI testing of smartphone applications: Case S60 and Linux. In Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman, editors, *Model-Based Testing for Embedded Systems*, Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, 2011. To appear.

[28] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[29] Scott McMaster and Atif M. Memon. Call-stack coverage for GUI test-suite reduction. *IEEE Trans. Softw. Eng.*, 2008.

[30] A. M. Memon. An Event-Flow Model of GUI-based Applications for Testing. *Software Testing Verification and Reliability*, 17(3):137–157, 2007.

[31] A. M. Memon, I. Banerjee, and A. Nagarajan. What Test Oracle Should I use for Effective GUI Testing? In *International Conference on Automated Software Engineering*, pages 164–173, 2003.

[32] Atif M. Memon. Advances in GUI testing. In Marvin V. Zelkowitz, editor, *Highly Dependable Software – Advances in Computers*, volume 58, pages 149–201. Academic Press, 2003.

[33] Atif M. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[34] Atif M. Memon. An event-flow model to test eds. In Enrique A. Belini, editor, *Software Engineering and Development*. Nova Science Publishers, 2009.

[35] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *WCRE*, pages 260–269, 2003.

[36] Atif M. Memon and Bao N. Nguyen. Advances in automated model-based system testing of software applications with a GUI front-end. In Marvin V. Zelkowitz, editor, *Advances in Computers*, volume 80, pages nnn–nnn. Academic Press, 2010.

[37] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a Goal-Driven Approach to Generate Test Cases for GUIs. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 257–266, New York, NY, USA, 1999. ACM.

[38] Atif M. Memon and Qing Xie. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31:884–896, October 2005.

[39] Bertrand Meyer. The Power of Abstraction, Reuse, and Simplicity: An Object-Oriented Library for Event-Driven Design. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods*, volume 2635 of *Lecture Notes in Computer Science*, pages 236–271. Springer Berlin / Heidelberg, 2004.

[40] G. J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 2 edition, 2004.

[41] A. Paiva, J. C. P. Faria, and R. F. A. M. Vidal. Towards the Integration of Visual and Formal Models for GUI Testing. In *Workshop on Model Based Testing*, pages 99–111, 2007.

[42] Ana Paiva, Nikolai Tillmann, João C. P. Faria, and Raul F. A. M. Vidal. Modeling and testing hierarchical guis. In *Proceedings of the 12th International Workshop on Abstract State Machines*, pages 329–344, 2005.

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

[44] João Carlos Silva, Carlos C. Silva, Rui D. Gonçalo, João Saraiva, and José Creissac Campos. The GUISurfer Tool: Towards a Language Independent Approach to Reverse Engineering GUI Code. In *2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 181–186, New York, NY, USA, 2010. ACM.

[45] Jaymie Strecker and Atif Memon. Relationships between Test Suites, Faults, and Fault Detection in GUI Testing. In *International Conference on Software Testing, Verification, and Validation*, pages 12–21, Washington, DC, USA, 2008. IEEE Computer Society.

[46] Jaymie Strecker and Atif M. Memon. Testing graphical user interfaces. In *Encyclopedia of Information Science and Technology, Second ed.* IGI Global, 2009.

[47] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[48] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test Input Generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29:97–107, July 2004.

[49] Lee White and Husain Almezen. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 110–121, Washington, DC, USA, 2000. IEEE Computer Society.

[50] Qing Xie and Atif M. Memon. Model-based testing of community-driven open-source gui applications. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 145–154, Washington, DC, USA, 2006. IEEE Computer Society.

[51] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1):4, 2007.

[52] Qing Xie and Atif M. Memon. Using a pilot study to derive a GUI model for automated testing. *ACM Trans. on Softw. Eng. and Method.*, 2008.

[53] Michal Young and Mauro Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

[54] Xun Yuan, Myra B. Cohen, and Atif M. Memon. GUI Interaction Testing: Incorporating Event Context. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010.

[55] Xun Yuan and Atif M. Memon. Using GUI Run-Time State as Feedback to Generate Test Cases. In *International Conference on Software Engineering (ICSE)*, pages 396–405, Washington, DC, USA, 2007. IEEE Computer Society.

[56] Xun Yuan and Atif M. Memon. Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback. *IEEE Transactions on Software Engineering*, 36:81–95, 2010.

[57] Xun Yuan and Atif M. Memon. Iterative execution-feedback model-directed GUI testing. *Inf. Softw. Technol.*, 52(5):559–575, 2010.