Behind the Scenes: An Approach to Incorporate Context in GUI Test Case Generation

Stephan Arlt*, Cristiano Bertolini†, Martin Schäf†

*University of Freiburg
arlt@informatik.uni-freiburg.de

†International Institute for Software Technology, United Nations University, Macau
{cbertolini, schaef}@iist.unu.edu

Abstract

Graphical user interfaces (GUIs) are a common way to interact with software. To ensure the quality of such software it is important to test the possible interactions with its user interface. GUI testing is a challenging task as they can allow, in general, infinitely many different sequences of interactions with the software. As it is only possible to test a limited amount of possible user interactions, it is crucial for the quality of GUI testing to identify relevant sequences and avoid improper ones.

In this paper we propose a model for better GUI testing. Our model is created based on two observations. It is a common case that different user interactions result in the execution of the same code fragments. That is, it is sufficient to test only interactions that execute different code fragments. Our second observation is that user interactions are context-sensitive. That is, the control flow that is taken in a program fragment handling a user interaction depends on the order of some preceding user interactions. We show that these observations are relevant in practice. We present a preliminary implementation that utilizes these observations for test case generation.

***Keywords*-GUI testing, grey-box testing, automated testing, model-based testing**

I. Introduction

Graphical User Interfaces (GUIs) are one of the most common way to interact with software systems. To ensure the functional correctness and robustness of such a system, special GUI testing strategies are applied. GUI testing can be considered as a special case of model-based testing. A model that represents possible sequences of user interactions with the GUI is generated or provided by the test designer. Based on this model, test cases can be generated. Using a model is necessary in order to avoid testing sequences of user interactions that are not possible in the actual system. Usually the process of generating and maintaining the model is a costly process that requires a lot of manual interaction. Every change to the GUI requires an adaption of the model and thus modifications to the test cases.

GUI testing automation is a common way to check stability and robustness of systems with GUIs. In GUI testing automation, the model of the GUI is generated from the source code of the application automatically. This model is used to generate sequences of user interactions. The effectiveness and efficiency of GUI testing automation depends on the underlying model that is used to generate sequences of user interactions. That is, how many test cases have to be generated to achieve a certain degree of confidence about the system and how many bugs can the test cases reveal?

As the GUIs become more and more complex, the total number of user interaction sequences that need to be tested is too large to be handled in a reasonable time [1]. It is impractical to test all necessary GUI sequences [2] even if the testing runs for a long time [3]. That is, there is an urgent need to improve the efficiency and effectiveness of the test case generation in GUI testing automation.

In this paper we propose a technique to generate a model for GUI testing automation. Our method is motivated by two observations:

- *Shared event handlers:* If some user interactions result in the execution of the same source code fragment, we say that this source code fragment is the *shared event handler* of these events. Generating sequences of user interactions that share event handlers will eventually result in redundant test cases.

- *Context-sensitive event handlers:* If the source code fragment of a user interaction contains a conditional choice that is evaluating a variable which has been modified in a preceding user interaction (e.g., the value of a check box), we say that this source code fragment is a *context-sensitive event handler*. In this scenario, the execution of a preceding event handler can affect the control flow of a succeeding event handler.

Based on these observations we present a way to generate a model for GUI testing that analyzes parts of the program's source code to detect sequences of user interaction that are suitable for testing. The presented technique uses a grey-box approach and performs a shallow analysis of those parts of the system that handle user interactions.

During the creation of the model, we identify if two user interactions use a *shared event handler*. Knowing that two events result in the execution of the same source code fragment allows us to avoid the creation of redundant sequences of user interactions during testing. This helps to increase the efficiency of our technique.

We use a shallow analysis to identify if the reaction of the system to a user interaction depends on input values that are influenced by other user interactions. That is, we try to identify if an event handler takes a conditional choice depending on variables that can be modified using other user interactions (in contrast to conditional choice that purely depends on internal data). We design our model such that it privileges the creation of sequences where events with context-sensitive event handlers are preceded by a random subset of events that form their context.

In Section III we present a new approach to create a model for GUI test case generation. Using our observations (Section IV), we utilize the source code of the event handlers to construct a model (Section V) that can generate efficient and effective test cases (Section VI). We apply our model to a medium-sized open source project. The results indicate that it is effective and efficient for real projects. We present a preliminary implementation and discuss the benefits and drawbacks of the presented model. Our experiments in Section VII show that shared event handlers and context-sensitive event handlers are relevant in practice.

The main contribution of this work is to show the utilizing of the source code of an application to generate more effective and more efficient GUI test cases. In particular, we detected a large amount of context-sensitive event handlers in our example applications. Thus, it motivates that analyzing the source code of GUI applications has a high potential to improve GUI test case generation.

II. Motivating Example

Our experiments were conducted using four applications. These applications are part of an open-source office (TerpOffice) suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. The TerpOffice applications are a common benchmark used by GUI testing and have been described in several studies [4].

We illustrate our approach using the TerpWord application as a running example. TerpWord is a simple word processor written in Java. Its GUI is implemented using the Java Swing toolkit [1].

TerpWord offers the standard functionality of a word processor. The user can interact with the application using buttons, top menus, context menus, etc., which we summarize using the term *widget*. Depending on the actions of the user, widgets might not be visible or accessible at a given moment. That is, the available user interactions and the shape of the visible user interface might change during the execution of the program. Figure 1 shows two screen shots of TerpWord where different widgets are enabled.

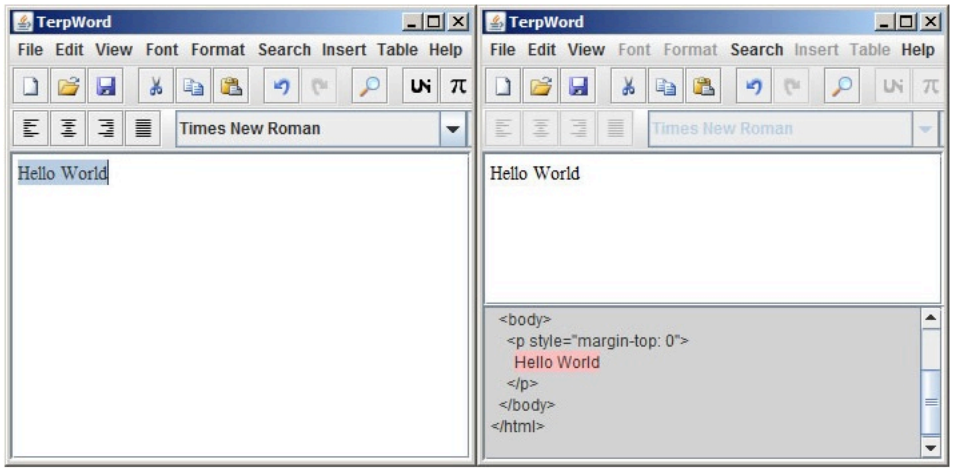
Even though TerpWord is a small application with a total of only 50 widgets in its main frame window, it is already challenging to test its user interface. Even if we only consider sequences of 3 consecutive different user interactions, we already have a total amount of 125,000 (50^3) possible test cases (without considering e.g., different input data or preconditions about the state of the system). Many of these test cases will not be feasible, as they do not correspond to possible user interactions with the application. Therefore it is crucial to find a model that helps to identify and generate feasible and relevant sequences of user interactions that can serve as test cases for the user interface of an application.

III. Grey-box GUI Testing Technique

In this section we give an overview of our GUI testing technique. We describe the approach in 3 steps: (i) mining the GUI source code; (ii) generate the model; and (iii) generate test cases. Figure 2 presents an overview of our approach.

An application is defined by a GUI and a set of instructions. The GUI is a set of *widgets*. Widgets can be buttons, text boxes, etc. They all share that the user can interact with them. For each interaction a widget generates an event e that carries information about the widget the user interacted with and the type of interaction. For each event e there exists a subset of instructions h in the source

[1]http://java.sun.com/javase/technologies/desktop/


TerpWord
File Edit View Font Format Search Insert Table Help
Times New Roman
Hello World
TerpWord
File Edit View Font Format Search Insert Table Help
Times New Roman
Hello World

```
<body>
  <p style="margin-top: 0">
    Hello World
  </p>
</body>
</html>
```

Figure 1. TerpWord with disabled and enabled Split Panel.

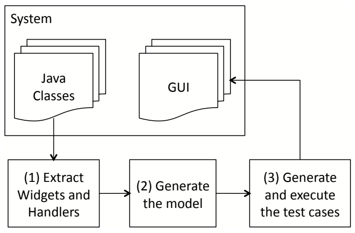

System
Java Classes
GUI
(1) Extract Widgets and Handlers
(2) Generate the model
(3) Generate and execute the test cases

Figure 2. GUI Tests Case Generation.

code that handles this event. We call h the event handler of e. The event handler h is a fragment of the application that is executed every time the associated event e occurs. We say that E is the set of all events and H is the set of instructions of the application (i.e., an event handler h is a subset of H). We assume that we have a relation $Ex : E \times H$ that allows us to identify the set of instructions $h = Ex(e)$ that handles an event e.

A *GUI test case* then is a sequence of events $t = e_1 \ldots e_n$ of user interactions and an oracle that decides if the outcome of this sequence meets some requirements. In this paper we limit our experiments to a simple crash oracle. That is, a test fails if the system terminates abnormally during the execution of the test case.

Our model for GUI testing is an automaton $M = (S, \delta)$, where S is a finite set of states, $\delta = S \times (E \cup \{\epsilon\}) \times S$ is a set of transitions between two states labeled with an event $e \in E$ or with ϵ. Every state in this automaton is accepting. That is, each word accepted by this automaton is a sequence of events that can serve as a GUI test.

The proposed technique generates the model from the source code of a given application automatically and generates a sequence of GUI test cases using this model. The *effectiveness* of our GUI testing technique is given by how many sequences have to be generated to test a certain fragment of H. The *efficiency* of a GUI testing technique is given by the ratio of the generated sequences that detect undesired behavior to the total number of generated sequences.

The model of the application is generated automatically by mining the application's source code. During the creation of the model we make use of the following two observations:

- *Shared event handlers*: Given two events e_1, e_2 and a program fragment $h \in H$. If $h = Ex(e_1) = Ex(e_2)$, we say that h is a shared event handler of e_1 and e_2, and it is sufficient to test either e_1 or e_2. That is, if two events result in the execution of the same part of the system $h \in H$, removing one of two events from the model might increase the efficiency of the generated test cases.
- *Context-sensitive event handlers*: Given an event e that is handled by a part of the system $h = h_1 \cup h_2$. We can think of $h_1 \cup h_2$ as a conditional choice between two different control-flow paths. If the choice whether h_1 or h_2 is executed when e is handled depends on some variables that are modified by another event e', we say that h is a context-sensitive event handler with respect to the context e'. In general, the context of an event handler might comprise several event handlers.

Notice that our observations are about facts that cannot be computed precisely. We can neither statically decide the events that use one shared event handler nor we can identify the possible context of an event handler for the general case. Our approach cannot provide a set of test cases that is minimal in one way or another. The focus of this work is to show that using these observations results in a model that generates a more efficient and more effective set of test cases than an approach that does not use these observations. In the following we show that our model can be used to generate a more efficient and more effective set of test cases than pure black-box models. We give evidence that our observations occur in practice and considering them during the model generation results in significant improvements in the test case generation.

For the construction of our model for GUI testing we mine information from the source code using a simple static analysis technique. We collect the set of events that will form our test sequences later on and the source code fragment that is executed when this event is handled (event handlers). During this step we identify *shared event handlers* and rule out those events that are handled by the same handler. The result of this step is a set of pairs that associates each event with one *unique* source code fragment that is executed when this event is handled. Then, we compute for each *(context-sensitive) event handler* the set of events that might influence its control-flow.

With the information mined from the source code about events and the context events of their respective event handlers, we create the automaton model that is used for test case generation.

IV. Mining Widgets and Event Handlers

We use the application TerpWord as a running example to explain our process of model generation for GUI testing. TerpWord is an instance of a standard windows application comparable to WordPad and Microsoft Word.

For generating the model of the GUI, we mine information about the possible user interactions from the source code. We analyze the application to collect all events (or user interactions) together with the source code fragment that handles this event. The mining procedure iterates over a Java program given as a set of *classes*. For each class we iterate over the *attributes*. For each attribute we check if the attribute is derived from the class `JComponent`, that is, if the attribute is an instance of a *widget type* that appears in the GUI. If the attribute is not a widget type we skip it and continue with the next attribute. If the attribute is a widget type, the algorithm iterates over the event handler associated with this widget. For the Java Swing toolkit, an event is registered using the method `setActionCommand`. We collect all events created for one widget. The corresponding event handler is usually a code block in a switch case, where the conditional is a string comparison with the event name. In our implementation we refer to this block as an event handler.

For demonstration purpose we restrict our presentation to the detection of shared event handlers in Java applications that implement their user interface using the Java Swing toolkit. Extending the approach to other patterns, toolkits and languages is straightforward.

Algorithm 1: Mining Widgets and Event Handlers

```
Input: Set of Java Classes
Output: Ex
1  begin
2    foreach (Java class in set) do
3      foreach (attribute in Java Class) do
4        if (attribute is a widget) then
5          foreach (Event e of widget) do
6            if (e is in Ex) then
7              Skip (redundant) widget
8            endif
9            else if (event handler h is empty)
               then
10             Skip (dead) widget
11           endif
12           else
13             Store Ex(e) = h
14           endif
15         endfch
16       endif
17       else
18         Skip attribute
19       endif
20     endfch
21   endfch
22 end
```

In the following we present Algorithm 1 that mines the widgets and event handlers of an application and allows us to identify shared event handlers.

A. Shared Event Handlers

While mining events and event handlers from the source code, Algorithm 1 identifies the use of shared event handlers. A shared event handler occurs when two different widgets offer user interactions that are registered with the same event name and the same action handler. Shared event handlers are a common practice to avoid duplicated functionality in the source code. However, for GUI testing only one of the events has to be considered.

The usage of a *shared event handler* is shown in Listing 1. The class constructor creates a menu item `m` (line 4) and a button `b` (line 8). Both created objects obtain exactly the same action 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     try {
17       String command = ae.getActionCommand();
18       if ( command.equals("textcopy") ){
19         if ( splitPane.isVisible() &&
20              sourcePane.hasFocus() ){
21           // copy text from source panel
22           sourcePane.copy();
23         }
24         else {
25           // copy text from main panel
26           mainPane.copy();
27         }
28       }
29       else if ( command.equals("splitpane") ){
30         // toggle split panel
31         splitPane.setVisible(
32           !splitPane.isVisible());
33       }
34     }
35   }
36 }
```

Listing 1. Example of a Shared Context-sensitive Event Handler in TerpWord.

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.

For each attribute in the source code that represents a widget, our algorithm records a pair of an event e that represents the interaction with this widget, and the program fragment h that handles this event. The fragment h is obtained by following the control-flow starting from the location that handles e. If we already recorded a pair (e', h) such that the events e and e' are handled by the same source code fragment h, we dismiss the pair (e, h), as the event e' is sufficient to test h. In the special case that h does not have control-flow (that is, the event is *not implemented*) we dismiss the pair (e, h) again. In the resulting set, each event e is associated with a unique source code fragment h.

For our example from Listing 1, Algorithm 1 returns the pairs:

$$\left(\textit{textcopy},\ \begin{array}{l} \texttt{if (splitPane.isVisible() \&\&} \\ \quad \texttt{sourcePane.hasFocus()) \{} \\ \quad \texttt{sourcePane.copy();} \\ \texttt{\}} \\ \texttt{else \{} \\ \quad \texttt{mainPane.copy();} \\ \texttt{\}} \end{array} \right),$$

$$\left(\textit{splitpane},\ \begin{array}{l} \texttt{splitPane.setVisible(} \\ \quad \texttt{!splitPane.isVisible()} \\ \texttt{);} \end{array} \right)$$

Now we have a set of pairs that associates each event with an event handler. The second part of our implementation identifies the context-sensitive event handlers and creates a model for test case generation.

B. Context-sensitive Event Handlers

From Algorithm 1 we obtain a list of pairs of events and event handlers. Then, we identify the *context* of each event handler. Given a pair (e, h) of event and handler, we say that the code h that handles an event e is a *context-sensitive event handler* if the control-flow path taken when e is handled depends on variables that are modified by other event handlers. That is, if the control-flow path taken in h is influenced by certain preceding events.

Algorithm 2: Collecting Context-sensitive Event Handlers

```
Input: Pair $(e, h)$ of event and handler
Output: Context events $Ctx$(h)
1 begin
2   $Ctx$(h)=$\emptyset$ ;
3   foreach (conditional choice $c$ in $h$) do
4     if ($c$ reads field of widget $w$) then
5       Add events of $w$ to $Ctx$(h);
6     endif
7   endfch
8 end
```

Algorithm 2 detects the context events $Ctx(h)$ for each event handler h. For each event handler h, it returns a list of events which we refer as a *context* of h. Initially the list of context events $Ctx(h)$ is empty. The algorithm traverses the control-flow of h. For each conditional choice (i.e., loops, if-then-else, etc.), the algorithm checks if the condition evaluates attributes of objects that are derived from a `JComponent`. If such an object is detected, we add the events associated with this widget to the list of context events $Ctx(h)$ of the handler h.

In the example given by Listing 1 of TerpWord the event handler `textcopy` evaluates if the split panel is visible (`splitPane.isVisible()`) and the source panel has the focus (`sourcePane.hasFocus()`) (line 19, 20). This conditional decides, whether the code of the source panel or the text of the main panel has to be copied to the clipboard. Both parts of the conditional refer to other widgets and thus can be modified by interaction with these widgets. Our previous analysis identified that the event *splitpane* is associated with the widget `splitPane` and that widget `sourcePane` is not associated with any event. We say that event handler of `textcopy` is context-sensitive with respect to the event `splitPane`. That is, we have a chance that, if we create different sequences of events (i.e. different test cases) where `textcopy` is preceded by the events associated with `splitPane`, we will execute the `then`-, and the `else`-branch of this conditional.

Our algorithm is primitive and does not consider data-flow. It uses a simple pattern matching. Still, as we discuss later, the algorithm is already able to detect a large amount of context-sensitive event handlers and their context events.

Now, that we have identified a set of events, event handlers and the context events for each event handler we can create our model that is used for the generation of GUI test cases.

V. Model Generation

From our preceding analysis we obtain a list of pairs (e, h) of event e and event handler h, as well as a list of events $Ctx(h)$ that might affect the control-flow of h. Using this information we create our automaton model that is used for test case generation.

Our model $M = (S, \delta)$ is a finite automaton with transitions labeled with events e or ϵ. A word accepted by the automaton is a sequence of events which represents a GUI test case. Initially our model contains an initial state $s_0 \in S$ and no transitions.

For each pair (e, h), we create a state s and one transition (s_0, e, s) and another transition (s, ϵ, s_0) that loops back to the initial state. That is, the automaton accepts any sequence of events.

Next we process the context events of (e, h). We iterate over the set of context events $Ctx(h)$. We create a new state s' and an edge (s_0, e_c, s') for each $e_c \in Ctx(h)$ and one edge (s', e, s). The additional states and edges do not affect the language accepted by the automaton (as it already accepts any possible sequence). Extending the automaton with these transitions is motivated by the way we create sequences of events for testing: If we traverse the model starting from the initial state s_0, and we always choose a random transition to the next state, the probability that an event e is preceded by an event $e' \in Ctx(Ex(e))$ is higher than the probability that e is preceded by some other event.

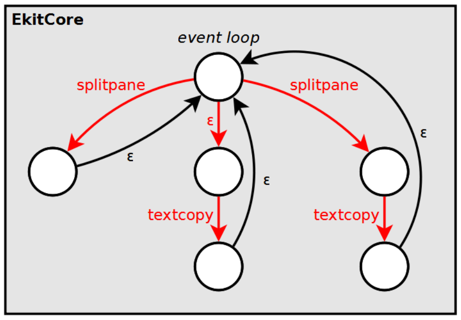


Figure 3. Example of a generated Model.

Figure 3 shows an example of the generated grey-box model for the TerpWord application. In two out of three paths in that automaton, the event $textcopy$ is preceded by the event $splitpane$. That is, if we generate sequences of events by randomly traversing through the automaton, it is more likely that the event $textcopy$ is preceded by $splitpane$.

VI. Test Case Generation

Algorithm 3 shows how we perform GUI testing based on the model generated in the previous sections. The input of the algorithm includes the model, *TCsize* that determines the number of events of each test case, *timeout* that determines for how long the test cases will be generated and the *seed* that determines the random seed used for test case generation.

Algorithm 3: GUI Test Case Generation

```
Input: Model $M = (S, \delta)$, TCsize, timeout, seed
1  begin
2      State $s$ = initial state of $M$
3      Test Case $tc := \{\}$
4      while (timeout is not reach) do
5          Pick randomly $(s, e, s') \in \delta$
6          $s := s'$
7          $tc := tc + e$
8          if (size of tc $\geq$ TCsize) $\wedge$ (s is initial state of $M$) then
9              try
10                 Execute the test case tc
11             catch
12                 Report $tc$
13             endtry
14             Test Case $tc := \{\}$
15         endif
16     endw
17 end
```

The algorithm starts with an empty sequence of events tc. The testing procedure is repeated until a given *timeout* is reached. Starting from the initial state in our model M, in each iteration of the loop, we randomly pick one outgoing edge and traverse to the target state. We add the event from the edge label and add it to tc (if the label is ϵ we do not add anything). This procedure is repeated until the initial state of M is reached again and the length of tc is larger than the threshold $TCsize$.

We execute the resulting sequence of events tc and check if the program terminates abnormally. If so, we report this sequence to the test engineer. Then, we reset the sequence of events and start over until the time limit is reached.

VII. Exploratory Experiments

We present our preliminary experiments using as subject four applications: TerpWord, TerpPaint, TerpPresent and TerpSpreadSheet. Our main goal is *to determine how the event handlers can affect the generated GUI test cases*. We define the following questions:

- *RQ1*: How often do shared event handlers occur?
- *RQ2*: How often do context-sensitive event handlers occur?

Table I presents how many shared event handlers and context-sensitive event handlers were found in four TerpOffice applications. We mine the source code of the applications looking for the number of widgets, event handlers, shared event handlers and context-sensitive event handlers. It is important to observe that we mine only the main frame windows and we are not considering non-main frame windows (e.g. modal windows like dialogs or modeless windows like floating widgets). According to the results we have the following observations:

- The mean of shared event handlers is 18%, which means that we have widgets that exercise the same event handler. The TerpPaint application is an outlier. If we are not considering this outlier the mean of shared event handlers is 27%.
- In the application TerpPaint we found only one shared event. The reason of that is due to its characteristics: it is an image manipulation application where almost all event handlers are only connected to one widget, since the application is supposed to be used mainly with one input device (e.g. a mouse that chooses a color from the palette and draws a rectangle). In comparison, the remaining TerpOffice applications tend to provide more than one input method (e.g. a mouse that clicks a toolbar button and a keyboard that chooses a menu item that leads to the execution of the same event).
- The mean of context-sensitive event handlers is 81%, which we consider high. The data used in the GUI seems to be important for test case generation. In TerpSpreadSheet almost all event handlers have an associated context.

Application: TerpOffice	#LOC	#Widgets	#Event handlers	#Shared event handlers	#Context-sensitive event handlers
Word	6842	50	39	11 (28%)	24 (62%)
Paint	17730	92	95	1 (1%)	69 (73%)
Present	25072	115	91	24 (26%)	74 (81%)
SpreadSheet	126909	51	37	10 (27%)	35 (95%)
Total (mean)	**176553**	**308**	**262**	**46 (18%)**	**213 (81%)**

Table I. Mining Widgets Evaluation.

A. Coverage: Shared Event Handlers

Table II presents the coverage of the test case generation for the main frame window of TerpWord *without* regard to shared and context-sensitive event handlers. We realize this test case generation by using a temporary model which is similar to the model presented in Section V, though neither shared nor context-sensitive transitions are considered here. Thus, for each shared event handler mined from the source code, we (1) add a transition to our temporary model, and (2) duplicate that transition, depending on how many widgets are connected to this event handler. This scenario conforms to a black-box perspective where *shared event handlers* are unknown. Additionally, for each context-sensitive event handler we (1) add a transition to our temporary model, but we (2) omit any corresponding ϵ or preceding transition.

We generate test cases randomly with a timeout of 1 min. The timeout comprises the time for starting and exiting the application itself, as well as the execution of the event handlers for each test case.

TCsize is the number of events (steps), *#Test Cases* is the total number of generated test cases, *ØMethod Cov.* is the percentage of methods coverage and *ØLine Cov.* is the percentage of lines covered.

Seed	TCsize	#Test Cases	ØMethod Cov. (%)	ØLine Cov. (%)
1	2	15	48	55
1	3	14	47	54
2	2	15	47	52
2	3	14	49	56
3	2	15	43	53
3	3	14	48	55
Avg.	**2**	**15**	**46.0**	**53.3**
Avg.	**3**	**14**	**48.0**	**55.0**

Table II. Coverage omitting Shared and Context-sensitive Event Handler, and a Timeout of 1 min.

Table III presents the coverage of the test case generation *with* regard to shared event handlers only. We realize this test case generation by using our model presented in Section V, though context-sensitive event handlers are not considered here. Thus, each shared event handler mined from the source code is represented by one transition only. Additionally, for each context-sensitive event handler we still do not insert their corresponding ϵ or preceding transitions in our model. Only Table IV shows the results for both shared and context-sensitive event handlers.

In this experiment, we also consider the main frame window of TerpWord and a timeout of 1 min. *TCsize* is the number of events (steps), *#Test Cases* is the total number of generated test cases, *ØMethod Cov.* is the percentage of methods coverage and *ØLine Cov.* is the percentage of lines covered.

Seed	TCsize	#Test Cases	ØMethod Cov. (%)	ØLine Cov. (%)
1	2	15	52	60
1	3	15	57	62
2	2	14	53	60
2	3	14	54	62
3	2	15	53	61
3	3	14	54	61
Avg.	**2**	**14.7**	**52.7**	**60.3**
Avg.	**3**	**14.3**	**55.0**	**61.7**

Table III. Coverage considering Shared Event Handler and a Timeout of 1 min.

According to the Tables II and III we can observe that we do not lose information when using shared event handlers. The coverage difference of both tables was about 7%. When we eliminate redundant events we increase the coverage. It seems that we can reach a higher coverage eliminating redundant event handlers in comparison with purely random test case generation. We believe that more and better results will come with bigger and more complex GUI applications, which have a lot of shared event handlers.

B. Coverage: Context-Sensitive Event Handlers

Table IV shows the coverage of the test case generation for the whole TerpWord application. In this table, both shared and context-sensitive event handlers are considered. Each test case has a size of 3 and the experiment ran 10 times with a timeout equals to 5 min.

We can observe that the coverage does not change significantly when we use different seeds for random test case generation based on our model. It means that the random selection of events has no influence in the results. The difference between the coverage found in this table with the previous one occurs because we are considering here the whole TerpWord application, and not only the main frame window.

Seed	Method (%)	Line (%)	#Test Cases
1	47	50	48
2	46	50	48
3	47	50	48
4	47	50	48
5	47	50	48
6	46	50	48
7	46	50	48
8	46	50	46
9	53	53	47
10	54	54	46
Avg.	**47.9**	**50.7**	**47.5**

Table IV. Coverage on TerpWord using our Technique and a Timeout of 10 min.

C. Discussion

The two main advantages of our technique are: the test case generation (i) with less redundant test cases and (ii) more interesting sequences of events. For (i) we identify the shared event handlers and we remove duplicated widgets. However, to identify the context is much more complicated. In our findings we observe that the applications do not follow code standards. For instance, event handlers are implemented within an *actionPerformed* method (e.g. in TerpWord), in a separated method or even in a different class. Thus, the mining algorithm has to be tailored to recognize specific patterns. In general, we think that the scalability of the static code analysis is strongly connected to the complexity of the patterns. It may be more difficult to mine context in the source code implementing sophisticated patterns. In terms of the scalability of the test case generation, we assume that our systematic approach is also useful for bigger applications, since the advantages of grey-boxing GUI applications are encoded in the model itself.

Related to our research question: for RQ1 we observe that shared event handlers occur often in the TerpOffice applications. However, we also found an application, which amount of shared event handlers is small (in TerpPaint we only found one event). For instance, in TerpPaint an event handler h_1 sets the color of a brush. Then, in an event handler h_2 the brush draws a pixel in the document using the selected color, without evaluating any other widgets. This behaviour can be found in many event handlers in TerpPaint.

For RQ2 we observe that the occurrence of context-sensitive event handlers are the most common event of the TerpOffice applications. It means that if we do not consider how to evaluate the context of these events probably our testing will not achieve a better coverage. This is one reason why GUI testing, in general, is used to assure reliability and robustness. When some context is incorporated into GUI testing we can achieve not only a high coverage, but improve the correctness of the applications. On the other hand, one limitation of our approach is related with the kind of context and different patterns, which our implementation deals with.

VIII. Related Work

Recent works have been done to improve GUI testing using the application's source code. Zhao and Cai [5] propose a coverage criteria based on the handler of the widgets. The idea is to apply GUI test cases and to verify how many handlers were exercised by the tests. Our approach extends the idea of handler-based coverage criterion to test case generation.

Some approaches to extract models from source code are proposed in [6], [7]. In [6] they extract the hierarchical structure of windows and interactive controls within windows and their properties. They consider enabling and disabling dependencies as well as data dependence. In [7] a tool called GUISurfer is presented, which automates the process of extracting information on the GUI behavior using its source code. Both approaches are complementary in comparison to our approach with respect to the kind of information they obtain in their reverse engineering.

Memon *et al.* [8] propose a novel model for GUI testing. On their findings, they define a single model that can be used by GUI and web-based applications. In [9] they propose a test case generation based on runtime execution. It allows identifying further user interactions between GUI events to increase the amount of detected faults. With respect to our model, we can also use the ideas of a single model to represent GUI and web-based applications. In this work, we do not analyze runtime executions for incorporating these interactions in our model, but it can be a future direction to identify different context.

Nguyen *et al.* [10] present a tool to map the business logic and presentation logic for GUI testing. It is interesting for applications with a business logic and multiple presentation logics like front-ends for email clients.

IX. Conclusion

In this paper we present a technique, which incorporates shared and context-sensitive event handlers into test case generation. We mine the source code of the application in order to identify shared and context-sensitive event handlers. We generate a model with this information and then we generate GUI test cases.

Our exploratory experiments show that simple static analysis already can mine useful information to avoid the creation of improper sequences of user interactions in GUI testing. We observe that about 18% of the total of event handlers are shared event handlers which suggests that we do not have to generate test cases for all widgets. In addition, we observe that context-sensitive event handlers represent the majority of the event handlers for the observed applications (about 81%). We are surprised that a simple pattern based analysis was already able to reveal so many dependencies between widgets. Incorporating these dependencies in the test case generation yields a large potential to improve the relevance of the test cases (and thus improve the quality of the application).

The results from our test case generation indicate that the benefit of eliminating redundant events handlers shows for applications where only a very limited number of sequences can be generated within the available time limit. Our model for test case generation seems to be improper to benefit from the information about context-sensitive event handlers. However, we believe that the knowledge about context events is useful and can be validated using a modified model.

A. Future Work

We see two potential directions for future work. One direction is to further mine information about the dependencies between user interactions. We believe that a more in-depth static analysis will be needed for significant improvements. Our next steps involve experiments with data-flow analysis and symbolic execution (e.g., [11]) to categorize the way in which an event handler modifies the data of its associated widget. Based on this we might be able to *guess* which order of context events might be suitable to trigger a certain control-flow path.

We plan to improve the kind of context that we are able to obtain from the code as well as to extend the focus to change-based GUI testing [12]. Changes in GUIs occur often even when the software is in production. The costs for testing the whole GUI are high. We will analyze how the GUIs change and how to check only for changes to aim a reduction of the testing costs.

Another direction of future work will be the creation of an improved model. From our experiments we could not see an effect of context-sensitive event handlers. We will conduct further experiments on larger applications as we assume that an effect is only visible for longer sequences of user interactions. We experiment with different ways to modify the model such that it creates certain sub-sequences of events with a higher probability than other. We believe that the approach of using a model that privileges certain sequences has a large potential in user interface testing, and can be additionally combined with e.g. machine learning approaches.

Our results and insights from detecting shared event handlers motivate the idea of further reducing the number of relevant events by, e.g. detecting widgets that are not associated with any code, or with code that does not change the state of the system (e.g., an info dialog). This analysis might seem trivial, but reducing the total number of events that need to be considered for test case generation has a huge impact if only a small fraction of sequences can be tested within the given time frame.

In the future we will perform a comparison with classical and well established techniques that use Event Flow Graphs as input for test case generation [13], [4], [2].

We think that the mining of hidden widgets can also be an improvement to GUI test case generation, respectively to achieve higher code coverage. For example, a hidden widget is a shortcut connected to a certain event handler, in the same manner as the event handler of a button's click. Since shortcuts that can be used by a user are not visible to them, mining the source code can reveal event handlers that are called, if a particular key stroke occurs. In comparison, in a back-box approach (e.g. GUI ripping) it might be much harder to detect those kinds of hidden widgets. So far, our mining algorithm searches for action listeners only, but we plan to consider further listeners for keys, a mouse and for windows itself.

In this paper, we only explored the widgets and event handlers of the main frame window of each application, but we want to extend our approach also to modal windows. Here the challenge is certainly to modify the grey-box model in a way, that a subset of possible event handlers can be enabled or disabled, according to whether a modal window is displayed or not.

Overall, we see that simple static analysis can reveal a lot of information about the dependencies between user interactions with a GUI. Using this information to distinguish between feasible and unfeasible test sequences has a great potential to generate good testing results even in a narrow time frame.

Acknowledgment

The authors would like to thank Volker Stolz for valuable comments. This work is supported by the research project ARV funded by the Macau Science and Technology Development Fund.

References

[1] L. White and H. Almezen, "Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences," in *Proceedings of the 11th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 110–121.

[2] A. M. Memon, "An Event-Flow Model of GUI-based Applications for Testing," *Software Testing Verification and Reliability*, vol. 17, no. 3, pp. 137–157, 2007.

[3] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 235–244.

[4] A. M. Memon and Q. Xie, "Studying the fault-detection effectiveness of gui test cases for rapidly evolving software," *IEEE Trans. Softw. Eng.*, vol. 31, pp. 884–896, October 2005.

[5] L. Zhao and K.-Y. Cai, "Event Handler-Based Coverage for GUI Testing," in *International Conference on Quality Software*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 326–331.

[6] A. Grilo, A. Paiva, and J. Faria, "Reverse Engineering of GUI Models for Testing," in *5th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE Computer Society, 2010, pp. 1 – 6.

[7] J. a. C. Silva, C. C. Silva, R. D. Gonçalo, J. a. Saraiva, and J. C. Campos, "The GUISurfer Tool: Towards a Language Independent Approach to Reverse Engineering GUI Code," in *2nd ACM SIGCHI symposium on Engineering interactive computing systems*. New York, NY, USA: ACM, 2010, pp. 181–186.

[8] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a Single Model and Test Prioritization Strategies for Event-Driven Software," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 48–64, 2011.

[9] X. Yuan and A. M. Memon, "Iterative execution-feedback model-directed GUI testing," *Inf. Softw. Technol.*, vol. 52, no. 5, pp. 559–575, 2010.

[10] D. H. Nguyen, P. Strooper, and J. G. Suess, "Model-Based Testing of Multiple GUI Variants Using the GUI Test Generator," in *AST '10: Proceedings of the 5th Workshop on Automation of Software Test*. New York, NY, USA: ACM, 2010, pp. 24–30.

[11] N. Tillmann and J. de Halleux, "Pex-White Box Test Generation for .NET," in *TAP*, ser. Lecture Notes in Computer Science, B. Beckert and R. Hähnle, Eds., vol. 4966. Springer, 2008, pp. 134–153.

[12] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change Bursts as Defect Predictors," in *21st IEEE International Symposium on Software Reliability Engineering (ISSRE)*, November 2010, pp. 309–318.

[13] 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*, 2003, pp. 164–173.