Grey-box GUI Testing: Efficient Generation of Event Sequences

Stephan Arlt
Institut für Informatik
Albert-Ludwigs-Universität
Freiburg
arlt@informatik.uni-freiburg.de

Cristiano Bertolini
Informatics Center
Federal University of
Pernambuco, Brazil
cbertolini@cin.ufpe.br

Martin Schäf
International Institute for
Software Technology
United Nations University,
Macau
schaef@iist.unu.edu

Ishan Banerjee
Dept. of Computer Science
University of Maryland
ishan@cs.umd.edu

Atif M. Memon
Dept. of Computer Science
University of Maryland
atif@cs.umd.edu

ABSTRACT

Graphical user interfaces (GUIs) encode, as event sequences, potentially unbounded ways to interact with software. During testing it becomes necessary to effectively sample the GUI's event space. Ideally, for increasing the efficiency and effectiveness of GUI testing, one would like to sample the GUI's event space by only generating sequences that (1) are allowed by the GUI's structure, and (2) chain together only those events that have data dependencies between their event handlers. We propose a new model, called an event-dependency graph (EDG) of the GUI that captures data dependencies between the code of event handlers. We develop a mapping between an EDG and an existing black-box model of the GUI's structure, called an event-flow graph (EFG). We automate the EDG construction in a tool that analyzes the bytecode of each event handler. We evaluate our "grey-box" approach using four open-source applications and compare it with the EFG approach. Our results show that using the EDG reduces the number of event sequences with respect to the EFG, while still achieving at least the same coverage. Furthermore, we are able to detect 2 new bugs in the subject applications.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; H.1.2 [**Models and Principles**]: User/Machine Systems

General Terms

Software Testing, System Testing;

ISSTA '2012 Minneapolis, Minnesota USA

Keywords

GUI Testing, Black-box, Grey-box, Test Automation;

1. INTRODUCTION

A particular challenge for system testing of software applications that have a graphical user interface (GUI) front-end is that the total number of all possible sequences of *user actions* is prohibitively large (in principle, possibly infinite), even for relatively small applications. A reasonably sized and effective sample needs to be selected for testing. GUI testing, i.e., system testing the software through its GUI is important, because most of today's software applications provide services to end-users via a GUI.

Each user interaction, e.g., pressing a key on the keyboard or clicking a mouse button, triggers an *event* in the application. An application responds to an event by executing a piece of code called the *event handler* associated with the event. In GUI testing, a *sequence of events* is an integral part of a GUI *test case*. In particular, a GUI test case consist of (1) a precondition that must hold before executing a sequence of events; (2) the actual sequence of events to be executed; (3) possible input-data to the GUI; and (4) the expected results of the test case (the oracle).

There has been extensive recent work on developing automated model-based GUI testing techniques. Current techniques (e.g., [4, 27, 10, 5, 28, 29, 12, 3]) use a *black-box* approach to generate test cases. Further, they use a graph-based model to represent the possible sequences of events with the GUI. Each node in these graph-based models represent an *event*, which is an interaction with one *widget* (e.g., selecting an element in a listbox). A path in this graph-based model corresponds to a sequence of events with the GUI; this sequence is used in the GUI test case.

In this paper, we propose and evaluate a *grey-box* [14] approach for automated GUI testing. The underlying mechanism for the grey-box approach is a new *event-dependency graph* (EDG) model that captures data dependencies between event-handlers in the GUI code. More specifically, an EDG is a weighted directed graph in which each node represents an event in the GUI. An edge from the node representing event e_1 to a node representing event e_2 shows that there is a *data dependency* from e_1's event handler to

e_2's event handler. The weight of the edge represents the number of fields that flow from e_1's event handler to e_2's event handler. Abstract event sequences are generated by using a minimax search [24] on the EDG. An abstract event sequence is a path through the EDG. Because of the nature of the EDG model, these abstract event sequence chain together only those events that have data dependencies between their event handlers. Further, an abstract event sequence does not necessarily mean that their events are allowed one after the other by the GUI's structure. For example, e_1 may be an event in the `MainWindow`, whereas event e_2 may be in the `FileOpen` dialog. An intermediate event that opens the `FileOpen` dialog is needed before e_2. Hence abstract event sequences, which are paths in the EDG, may not be executable, which is why we called them "abstract" event sequences above. To convert abstract event sequences into "executable" event sequences, a mapping is maintained between the EDG and the GUI's workflow, represented using an existing *event-flow graph* (EFG) black-box model of the GUI [20]. After applying the mapping, we obtain event sequences that (1) are allowed by the GUI's structure, and (2) chain together only those events that have data dependencies between their event handlers. By embedding these executable event sequences into GUI test cases, a compact test suite is formed, which efficiently samples GUI event space.

We evaluate the grey-box approach on four open-source applications: *TerpWord*, *Rachota*, *FreeMind* and *JabRef*. The results show a dramatic increase in the efficiency of the event sequence generation and execution. Further, one new bug in Rachota, and one bug in JabRef is revealed.

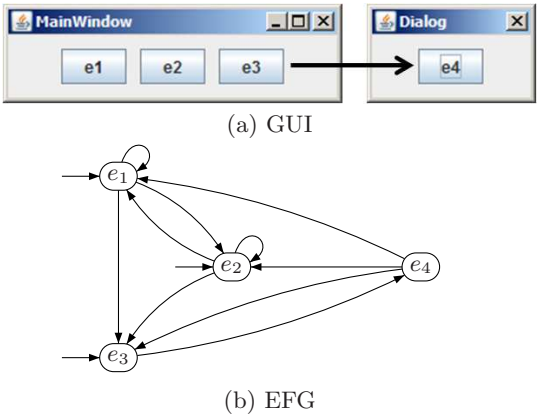
The paper is organized as follows: Section 2 provides the background of model-based GUI testing using a black-box approach. Section 3 introduces our grey-box GUI testing approach, which incorporates an event-flow graph (EFG) and an event-dependency graph (EDG) to generate efficient event sequences. Section 4 provides an overview of the implementation, which we use to evaluate the approach (Sections 5 through 6). Section 8 summarizes the related work, and finally, Section 9 presents the conclusions and future work.

2. BACKGROUND

When testing a system through its GUI, only a finite set of user interactions can be tested. The choice of this set is vital to the success of the testing procedure. A common way to sample the possibly infinite set of sequences is to use a graph-based model of the GUI, called *event-flow graph* (EFG).

An event-flow graph, $EFG = \langle E, I, \delta \rangle$, for an application is a directed graph. Each node $e \in E$ is an event in the GUI. An event is a response of the system to a user interaction (a click on a button triggers an `onClick` event). Each event in $I \subseteq E$ is an initial event which can be executed directly after the application launched. An edge $(e, e') \in \delta$ between to events $e, e' \in E$ states that the event e' can be executed immediately after the event e. Conversely, if there is no edge between events e, e' then event e' cannot be executed immediately after event e. This may be owing to structural characteristics of the GUI. For example, executing e may close the window containing e'. The EFG can be obtained automatically from the application using a *GUI Ripper* [21]. Section 4.2 outlines the construction of the EFG, its benefits and limitations.

Figure 1(a) shows the GUI of an example application. The `MainWindow` appears when the application is launched. A modal dialog `Dialog` appears when the button e_3 is clicked. It is closed when the button e_4 is clicked.



(a) GUI

(b) EFG

Figure 1: An Example Application.

Figure 1(b) shows the corresponding EFG of the example application, which consists of 4 events (e_1 to e_4), where the events e_1, e_2, e_3 represent initial events. The execution event e_3 opens the modal dialog, s.t. e_4 becomes accessible. The event e_4 closes the `Dialog` and thus, after e_4 is executed, it becomes inaccessible again.

An *event sequence* in an EFG is a sequence of events which represents a sequence of user interactions with the GUI. An executable event sequence $s = e_0, \ldots, e_n$ is a sequence of events which starts with an initial event $e_0 \in I$.

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

From the EFG, sequences of events of a particular length are sampled. For instance, using a sequence length of 1 leads to the following event sequences: $s_1 = \langle \mathbf{e_1} \rangle$, $s_2 = \langle \mathbf{e_2} \rangle$, $s_3 = \langle \mathbf{e_3} \rangle$, and $s_4 = \langle e_3, \mathbf{e_4} \rangle$. Note that sequence s_4 has length 2. This is because e_4 cannot be tested with a sequence of length 1, therefore additional *reaching steps* are introduced to connect e_4 to an initial event of the EFG.

Although event sequences of length 1 provide a compact set, it is certainly not sufficient for bug detection, since pairs and triples of events are not considered. However, increasing the length of the event sequences does not scale, as the number of generated sequences grows exponentially (Table 1 shows all event sequences generated with a length of 2 for the EFG in Figure 1(b)). That is, a better technique for sampling the EFG is needed in order to generate event sequences with a reasonable length. In the following we present a technique to efficiently generate a compact set of relevant event sequences of arbitrary length.

3. GREY-BOX GUI TESTING

An EFG is useful to generate feasible event sequences. However, when generating longer event sequences the number sequences becomes prohibitively large and a more sophisticated sampling strategy is needed.

$s_1 = \langle \mathbf{e_1}, \mathbf{e_1} \rangle$ $s_5 = \langle \mathbf{e_2}, \mathbf{e_2} \rangle$ $s_9 = \langle e_3, \mathbf{e_4}, \mathbf{e_2} \rangle$
$s_2 = \langle \mathbf{e_1}, \mathbf{e_2} \rangle$ $s_6 = \langle \mathbf{e_2}, \mathbf{e_3} \rangle$ $s_{10} = \langle e_3, \mathbf{e_4}, \mathbf{e_3} \rangle$
$s_3 = \langle \mathbf{e_1}, \mathbf{e_3} \rangle$ $s_7 = \langle \mathbf{e_3}, \mathbf{e_4} \rangle$
$s_4 = \langle \mathbf{e_2}, \mathbf{e_1} \rangle$ $s_8 = \langle e_3, \mathbf{e_4}, \mathbf{e_1} \rangle$

Table 1: Generated Event Sequences using an EFG Sequences Length of 2

To efficiently sample the event sequences generated from an EFG, we propose to incorporate additional information from the source code of the event handlers. Knowing which fields are modified and which are read upon the execution of an event makes it possible to prioritize sequences of events where the event handlers influence each other and to avoid those sequences, where events are completely independent (e.g., a `Copy` and a `Help` button in a word processor).

```
1  class MainWindow {
2    boolean enabled = true;
3    String text = "Hello World";
4
5    void e1() {
6      enabled = false;
7    }
8
9    void e2() {
10     text = text.toLowerCase();
11   }
12
13   void e3() {
14     if ( enabled )
15       openDialog(this);
16     else
17       Log.write(text);
18   }
19 }
20
21 class Dialog {
22   MainWindow mainWindow;
23
24   void e4() {
25     mainWindow.text = null;
26     closeDialog();
27   }
28 }
```

Listing 1: Java Snippet of the Example Event Handlers.

Listing 1 shows the Java snippet of the example application, especially of their 4 event handlers. The example application consists of the classes `MainWindow` and `Dialog`, where `MainWindow` contains three event handlers (`e1`, `e2`, and `e3`), and `Dialog` the event handler `e4`. Event handler `e1` sets the field `enabled` to `false`, and `e2` converts the string of field `text` to lower case. In `e3`, the field `enabled` is evaluated in a conditional. If `enabled` is `true`, the dialog is opened and the current instance `this` of `MainWindow` is passed to the dialog. If `enabled` is false, the content of `text` is written to a log. Event handler `e4` sets the field `text` of the current instance of `MainWindow` to `null` and closes the dialog.

The execution of the event sequence $\langle e_3, e_4, e_2 \rangle$ throws a `NullPointerException`, because the field `text` in `e2` was set to `null` in `e4`. This example application is a simplified version of a bug which we found in real world applications.

Without considering the application's source code, in the worst case, all sequences of length 2 must be generated and executed to detect the bug. For our example, this leads to 10 event sequences in total. When analyzing the source code of an application, we observe that certain event handlers share a *data dependency*, which helps to prefer or to neglect certain events from event sequence generation: Event `e1` writes field `enabled` which is read in `e3`; `e4` writes field `text`, which is read in `e2`. Further, there is no data dependency between e_1 and e_2. To utilize these data dependencies for a more efficient event sequence generation, we introduce a new graph-based mode called event-dependency graph (EDG).

3.1 Event-dependency Graph

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 (w) 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, we recursively follow potential method calls, collect these fields, and place 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 cause 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 events, and succeeding events on a path in the EDG might not be directly executable in the GUI, we refer to an EDG path as an abstract event sequence.

Definition 2. *Given an event-dependency graph $EDG = \langle E, \psi \rangle$. An abstract event sequence is a 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 1: 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       end if
12     end foreach
13   end foreach
14 end
```

Algorithm 1 shows how the EDG is constructed. The

algorithm takes the program P and a 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, we copy E to E' (line 2). Then, we iterate over all pairs of events e, e' (line 4).

We call the method `getFieldsWritten` which returns a set W of all fields that are written during the execution of the event handler of e (line 5). Then, we call the method `getFieldsRead` 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), we add a new edge to the edge which is labeled with the size of the intersection (line 10). Note that our 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).

3.2 Event Sequence Generation

Our event sequence generation is built out of two consecutive steps. First, we select potentially interesting sequences of events, called abstract event sequence, from the EDG using Algorithm 2. Second, we use the abstract event sequences to generate executable event sequences from the EFG using Algorithm 3.

Algorithm 2 takes an EDG and two parameters as input: *len* gives the maximum length of the abstract event sequence to be generated, and *top* gives the maximum number for abstract event sequences to be generated for each event. The algorithm returns a set Π of abstract event sequences. These are later used for generating executable event sequences.

Algorithm 2: Generating abstract event sequences.

```
   Input: ⟨E, ψ⟩ : Event-dependency graph,
          len : max length of abstract event sequence,
          top : max number of abstract event sequences per event
   Output: Π : set of abstract event sequences
 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             end while
12             if π ∈ Π then break
13             Π' = Π' ∪ {π}
14         end while
15         Π = Π ∪ Π'
16     end foreach
17     return Π
18 end
```

For each event $e \in E$, a new set Π' of abstract event sequences is created, which initially is empty (line 4). As long as the size of this set is smaller than *top* (line 5), we add further abstract event sequences (line 6). Each such abstract event sequence π initially contains only e (as we are looking for sequences of events that start in e). While the length of π is smaller than *len* (line 8), and the last event of π still has successors, the method `bestSucc` is finds the best possible successor and adds it to the end of π (line 10). The method `bestSucc` uses a *minimax* strategy to identify the best successor, unless this successor leads us on a path which is already in Π'. In this case, `bestSucc` returns the second best choice. We use the *minimax* strategy to minimize the selection of events with low dependencies.

The loop in line 5 terminates either, if it has collected *top* abstract event sequences 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 event sequence in Π, we want to generate an executable event sequence. However, the abstract event sequences are not necessarily executable, as consecutive events in the EDG might have no direct connection in the EFG. Therefore, we use Algorithm 3 to find one EFG path for each of these abstract event sequences, 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 the execution of two events. In that case, we split the sequence into two sequences that later on are tested immediately after each other.

Algorithm 3: Conversion from abstract event sequences to executable event sequences.

```
   Input: ⟨E, I, δ⟩ : Event-flow graph,
          Π : set of abstract event sequences
   Output: S : Set of executable event sequences
 1 begin
 2     Sequences of events S = {}
 3     foreach Sequence e_i, ..., e_j in Π do
 4         pick e_0 from I
 5         Path s = shortestPath(e_0, e_i)
 6         for k = i to j − 1 do
 7             s = s • shortestPath(e_k, e_{k+1})
 8         end for
 9         S = S ∪ {s}
10     end foreach
11     return S
12 end
```

Algorithm 3 takes an EFG and the set Π of abstract event sequences computed by Algorithm 2 as input, and returns a set of executable event sequences, which are paths in the EFG and start in an initial event. For each sequence of events $e_i \dots e_j$ in Π (line 3), a path s (line 5) is created. We pick the shortest path from an event $e_0 \in I$ to e_i, and then iterate over the events in the abstract event sequences and always add the shortest path between succeeding events to s (line 7). Then we add s to the set S (line 9). Since paths in S start in an initial event of the EFG, it can immediately executed as a GUI event sequence.

Infeasible event sequences are only generated if the EFG is not complete (e.g., because it was generated automatically) or if the data dependency analysis is imprecise. As these are implementation issues, we refer to Section 4 for details.

Figure 2 shows the EDG of our example application. If we apply Algorithms 2 with $len = 2$ and $top = \infty$, Algorithm 3 outputs the following executable event sequences: e_1 writes into e_3, which results in $s_1 = \langle \mathbf{e_1}, \mathbf{e_3} \rangle$. Since the field `text` is both read and written in e_2, $s_2 = \langle \mathbf{e_2}, \mathbf{e_2} \rangle$ is generated. e_3 does not write into any other event, and thus, is considered in a single event sequence $s_3 = \langle \mathbf{e_3} \rangle$. Finally, e_4 writes into e_2, which leads to $s_4 = \langle e_3, \mathbf{e_4}, \mathbf{e_2} \rangle$. Because e_4 does

not represent an initial event, the intermediate event e_3 is inserted.

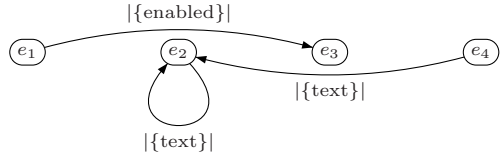


Figure 2: EDG of the Example Application.

Note that it is not possible to combine EFG and an EDG into one graph-based model: On the one hand, it is possible to label a directed edge (e_1, e_2) in the EFG with the weight of the data dependency (e.g., zero in case of a non-dependency). On the other hand, a directed weighted edge (e_3, e_4) has to be added to the EFG, if a data dependency is detected. However, the added edge may represent an event sequence, which is not allowed in the GUI.

4. IMPLEMENTATION

We integrate an implementation of the grey-box approach into GUITAR[1], which is a open source, model-based system for automated GUI testing. Figure 3 presents an overview of the GUITAR system. The grey-highlighted steps in the overview emphasize our extensions made to the GUITAR system. Considering the grey-box approach, testing an application using the GUITAR system consists of the following steps:

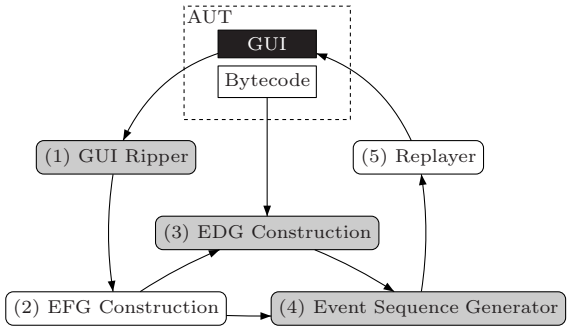


Figure 3: Overview of the GUITAR System, including the Grey-box Extensions.

4.1 GUI Ripper

In the first step, the *GUI Ripper* executes the AUT and records the GUI structure. A GUI structure consists of widgets (e.g., windows, buttons, and text fields) and their corresponding properties (e.g., enabled/disabled, height, and width). While executing the AUT, the GUI ripper enumerates all widgets of the main window using reflection, and stores the obtained information in the GUI structure. For each found widget (e.g., a button), GUI ripper triggers the assigned event (i.e., a button click). For instance, if the click on the button opens a new window, GUI ripper continues to record the GUI structure of that recently opened window and so on. The process stops, if all found windows have been explored. Since each GUI represents a hierarchical structure, a depth-first search is performed on the AUT's GUI. For the grey-box approach, we enhanced the GUI ripper, such that, for each widget the event handlers assigned to this widget are additionally stored in the GUI structure. This information is needed during the analysis of the bytecode which is performed as a part of the EDG construction.

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

4.2 EFG Construction

The GUI structure recorded by the Ripper serves as input to the *EFG Construction*, which automatically constructs the EFG that is used for the test case generation. While the GUI structure contains information about widgets and their properties, the EFG represents an abstract view which only contains the events and their following events. The EFG construction iterates over all windows in the GUI structure and creates a single EFG for each window. Later, these single 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 for each containing widget. 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 can not be triggered if the window appears. For each window an EFG is created and the event which opens/accesses other window is connected with all initial events of this other window. The details of the EFG construction can be found in [21].

4.2.1 Short Assessment of the EFG Construction

Since the GUI ripper performs a dynamic analysis of the GUI, it cannot be guaranteed to find all widgets of the AUT [20]. For instance, the AUT itself might be hostile or even faulty, e.g., if the GUI opens a new window in the background, the GUI ripper will not be able to find it, and thus, it cannot be considered during EFG construction. Further, the fact if a widget is enabled or disabled during ripping may strongly depends on the environment (e.g., user settings). These problems tend to be of technical nature and their severity might differ depending on the used platform.

Instead of executing the AUT in GUI ripping, it is in general possible to create the GUI structure and the EFG respectively via static analysis. However, a static analysis technique must be tailored to comprehend how a GUI is created. While there exist different code styles for creating GUI's, a static technique might find its limitations even if a GUI is defined outside the source code of the application, e.g., in XML files.

Note that the EFG of an AUT is not complete and represents an approximation of the AUT's event-flow. It cannot be guaranteed that a path in the constructed EFG is actually executable on the AUT's GUI. For instance, if a click on a button changes the entire parent window (e.g., removing or adding widgets), then the GUI ripper and the EFG construction respectively does not recognize these changes made to the GUI. A test engineer has to improve manually the EFG according to the actual behavior of the AUT.

4.3 EDG Construction

In order to construct an EDG, we perform a shallow bytecode [16] analysis of the AUT to obtain data dependency between events. In particular, the bytecode analysis records, which fields are read and written by each event handler, that

is, the functions `getFieldsRead` and `getFieldsWritten` in Algorithm 1. Hence, the Java bytecode and the constructed EFG of the AUT serve as input to the *EDG Construction*. For our bytecode analysis, we use the the ASM framework[2]. Other frameworks such as Soot[3] could be used equally well.

4.3.1 Bytecode Analysis

Listing 2 shows the bytecode of the event handler `e1` and `e4` from the example application in Listing 1. In bytecode, fields are read by the instruction `GETFIELD`, and written by `PUTFIELD`. Further, methods are called using the `INVOKE`[4] instruction. In line 2, a constant value of 0 is first pushed to the stack, and then assigned to field `enabled` in line 3. In line 6 and 7 respectively, field `mainWindow` and a constant value of `null` are pushed to the stack. Field `text` of `mainWindow` is then assigned with the value `null`. Finally in line 9, method `closeDialog` of is called.

```
1 void e1()V
2  ICONST_0
3  PUTFIELD MainWindow.enabled : Z
4
5 void e4()V
6  GETFIELD Dialog.mainWindow : LMainWindow;
7  ACONST_NULL
8  PUTFIELD MainWindow.text : Ljava/lang/String;
9  INVOKEVIRTUAL Dialog.closeDialog()V
```

Listing 2: Bytecode Snippet of the Example Event Handlers.

The EDG construction is preceded by one step: the creation of a class database (ClassDB). The ClassDB models the dependencies between fields, methods and classes of the AUT. During EDG construction, a request to the ClassDB determines the data dependency of two given event handlers. Figure 4 shows the ER model of the ClassDB.

In order to build a ClassDB, the bytecode analysis starts with visiting all classes of the AUT, since classes contain both methods and fields. In our implementation, it is possible to provide a *scope* (a set of JAR archives) to restrict the set of classes to be analyzed. For instance, only application classes are supposed to analyze and third-party libraries are discarded. Each class is stored in table `Class` of the ClassDB and is identified by its fully-qualified name, to avoid collisions if a certain class name is multiply used.

Then, the bytecode analysis visits all methods of each class. Note that it is important to inspect all methods, and not only those which are declared as event handlers. Moreover, it is necessary to follow all methods calls in each method, which can be detected by visiting the `INVOKE` instructions of the bytecode. For instance, method `e4` in Listing 2 calls method `closeDialog`, which may write further fields. Thus, there exist a recursive relationship *calls* between methods. Each method is stored in table `Method` in the ClassDB and is associated to its class.

For each method, the bytecode analysis fetches all fields that are read and written. This is can be detected by visiting the `GETSTATIC` and `PUTSTATIC` instructions of the bytecode. Read and written fields are stored in table `Field`, where each field is associated to its method.

[2]http://asm.ow2.org/
[3]http://www.sable.mcgill.ca/soot/
[4]`INVOKEVIRTUAL`, `INVOKESTATIC`, `INVOKESPECIAL`

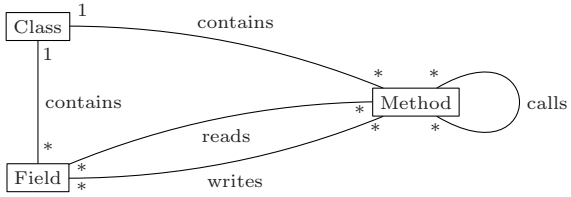


Figure 4: Simplified ER Model of the ClassDB.

Once all classes, methods and fields are visited and mapped in the ClassDB, Algorithm 1 uses 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, a possible data dependency between events is captured. Further, due to this shallow analysis of the bytecode, the computation time for building the ClassDB is low, even for big applications.

4.3.2 Short Assessment of the EDG Construction

Java distinguishes between instance fields and class fields, which are treated the same way in our bytecode analysis. 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 calls of instance methods and class methods and thus, is not reliable regarding polymorphism.

The bytecode analysis does not consider potential aliasing of fields or potentially infeasible control-flow. Hence, the resulting EDG is only an approximation of the actual data dependencies between fields. However, we are interested in prioritizing events, so a *cheap* bytecode analysis in terms of computation time is sufficient, while leaving room for further in-depth analyses.

4.4 Event Sequence Generator

The Event Sequence Generator takes as input an EFG and an EDG from the application. In this step, the Algorithms 2 and 3 are applied. The output is a set of executable event sequences, where each executable event sequence is embedded into one GUI test case.

4.5 Replayer

The Replayer is responsible for executing GUI test cases. A test case is considered as a precondition, an executable event sequence, input-data and an oracle. Figure 5 presents an overview of the Replayer process. It consists of the following steps: (1) it selects an executable event sequence; (2) it prepares a test case, which ensures that the precondition of the test case holds; (3) it executes the test case on the AUT, which performs the executable event sequence; (4) it restarts the AUT, which covers the events `exit` and `launch` of the AUT; (5) it evaluates, whether the test case has failed or passed.

5. EXPERIMENT

We compare our grey-box approach with the black-box approach by studying efficiency and effectiveness. *Efficiency*

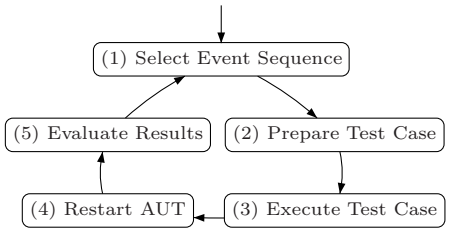


Figure 5: Overview of the Replayer Process.

is considered as the computation time for generating the abstract event sequences(in minutes) and the time for test case execution (in hours). *Effectiveness* is considered as the line and branch coverage (in percentage). We define two research questions. **Q1**: Is the grey-box approach efficient in terms of mean time to execute the test cases? And **Q2**: Is the grey-box approach effective in terms of mean code-coverage?

5.1 Setup of the Experiment

We evaluate the grey-box approach using four Java-based open source applications: *TerpWord 4.0* is a word processor, *Rachota 2.3* is a time recording system. *FreeMind 0.9.0* creates mind maps, and *JabRef 2.7* manages bibliographic references. It is important to observe that we use stable versions of all applications where bugs are rarely found. We choose these applications to consider both small and large applications (in terms of # of classes), and to cover different code styles. Table 2 shows some relevant statistics of the Applications Under Test (AUTs): the number of lines of code (**LOC**), number of classes (**Classes**), number of GUI events (**Events**), number of edges in the EFG (**EFG edges**), and number of edges in the EDG (**EDG edges**).

	TerpWord	Rachota	FreeMind	JabRef
LOC	6,842	13,750	40,922	68,468
Classes	215	468	1,362	4,027
Events	159	154	959	776
EFG edges	4,229	1,493	105,986	100,211
EDG edges	4,100	2,172	25,248	10,034

Table 2: Experiment setup.

Table 3 shows six different configuration used to test the four applications. For brevity of exposure we use identifiers (**ID**) to refer to these configurations.

The black-box approach presented in [20] is used in Configuration A, B, and C. These configurations are the baseline of our experiment. Configuration A generates one event sequence (length = 1) for each event in the EFG. Configuration B (length = 2) generates event sequences for each *pair* of events (e_i, e_j), that have a direct connection in the EFG. Configuration C (length = 3) generates event sequences for each *triple* of events (e_i, e_j, e_k), where $\{e_i, e_j\}$ and $\{e_j, e_k\}$ are direct neighbors in the EFG.

The grey-box approach is used in Configuration D, E. Configuration D considers abstract event sequences of length 2 and does not limit the number of abstract event sequences generated per event. Configurations E and F have abstract event sequences of length 3. Here, the number of generated abstract event sequences is limited to 50 and 100 respectively. This is because each event in the applications *TerpWord* and *FreeMind* has about 25 dependent events. That is, the number of abstract event sequences of this size is already large. In particular, we are interested in knowing, if doubling the number of generated abstract event sequences from 50 to 100 will have a significant impact on the coverage.

Black-box Approach		Grey-box Approach	
ID	**Configuration**	**ID**	**Configuration**
A	$len = 1$	**D**	$len = 2$
B	$len = 2$	**E**	$len = 3$; $top = 50$
C	$len = 3$	**F**	$len = 3$; $top = 100$

Table 3: Experiment configurations.

Each generated executable event sequence is embedded in a GUI test case, which is executed by the Replayer. The precondition of each test case states, that all user-settings have to be deleted before performing the event sequence on the AUT. As an oracle, a crash monitor is used, which records any exception found during the test case execution.

The test cases are executed on 10 Linux machines with a 4 x 2.0 GHz CPU, 4 GB RAM, 500 GB HDD. The experiment was executed two times with the same setup (e.g., the same seed for input-data) to ensure that the obtained results are reproducible. The total number of test cases executed is 6,089,403.

5.2 Experimental Results

Table 4 shows a summary of the experimental results. For each configuration we report the number of event sequences (*# es*), broken event sequences (*# broken es*), total generation time (*gen t (m)*), generation time per event sequence (*gen t per es (s)*), total execution time (*exec t (h)*), execution time per test case (*exec t per tc (s)*), line coverage (*line cov.*) and branch coverage (*branch cov.*). The event sequence generation time is expressed in minutes and the test case execution time in hours. The generation time per event sequence and execution time per test case are expressed in seconds.

Observation 1: Configuration A has the smallest number of event sequences and the lowest coverage. The number of event sequences corresponds to the number of events, which means that each event is only tested once. However, we believe that this approach is useful for smoke tests [23], since the generation and execution time is also the lowest amongst all the configurations.

Observation 2: As expected, Configuration D is significantly more efficient than Configuration B on the applications TerpWord, FreeMind, and JabRef. For these applications, both configurations have the same line and branch coverage. However, Configuration D uses significantly fewer event sequences and consumes less time than Configuration B. Thus, the grey-box approach generates more *efficient* event sequences than the black-box approach for these three applications.

Observation 3: For Rachota, Configuration D attains a higher coverage than Configuration B. However, more event sequences are generated in Configuration D, and the execution consumes more time. This is likely owing to the fact that in Rachota, the EDG has significantly more edges than the EFG.

Observation 4: The number of generated event sequences of JabRef in Configuration C exceeds 5 million. Comparing to Configuration B, the obtained coverage for JabRef is

	TerpWord	Rachota	FreeMind	JabRef
Configuration A (Black-Box Approach)				
# es	159	154	959	776
# broken es	0	0	5	5
gen t (m)	0.3	0.28	1.10	1.08
gen t per es (s)	0.12	0.12	0.12	0.12
exec t (h)	0.50	0.58	4.58	4.22
exec t per tc (s)	12	15	30	28
line cov. (%)	41	60	50	51
branch cov. (%)	22	31	36	22
Configuration B (Black-Box Approach)				
# es	3,307	1,310	11,396	43,017
# broken es	0	0	57	258
gen t (m)	6.62	2.62	24.68	93.2
gen t per es (s)	0.12	0.12	0.13	0.13
exec t (h)	11.94	5.82	98.13	358.48
exec t per tc (s)	13	16	31	30
line cov. (%)	55	61	53	54
branch cov. (%)	36	34	37	26
Configuration C (Black-Box Approach)				
# es	79,949	20,221	489,250	5,360,366
# broken es	0	0	2,446	32,162
gen t (m)	159.90	40.44	1,223.13	15,187.70
gen t per es (s)	0.12	0.12	0.15	0.17
exec t (h)	310.92	95.49	4,348.89	44,669.72
exec t per tc (s)	14	17	32	30
line cov. (%)	55	62	53	55
branch cov. (%)	36	36	38	27
Configuration D (Grey-Box Approach)				
# es	2,695	1,407	9,944	5,860
# broken es	0	0	63	83
gen t (m)	7.63	4.22	43.08	20.52
gen t per es (s)	0.17	0.18	0.26	0.21
exec t (h)	9.73	6.25	88.39	48.83
exec t per tc (s)	13	16	32	30
line cov. (%)	55	62	53	54
branch cov. (%)	36	36	37	26
Configuration E (Grey-Box Approach)				
# es	2,068	1,781	7,113	9,595
# broken es	0	0	45	135
gen t (m)	6.55	5.93	35.57	36.78
gen t per es (s)	0.19	0.2	0.3	0.23
exec t (h)	9.19	8.91	65.20	90.62
exec t per tc (s)	16	18	33	34
line cov. (%)	47	62	53	55
branch cov. (%)	26	36	38	27
Configuration F (Grey-Box Approach)				
# es	4,036	3,307	12,904	18,497
# broken es	0	0	81	261
gen t (m)	13.45	11.57	68.82	77.07
gen t per es (s)	0.2	0.21	0.32	0.25
exec t (h)	17.94	17.45	118.29	179.83
exec t per tc (s)	16	19	33	35
line cov. (%)	55	62	53	55
branch cov. (%)	36	36	38	27

Table 4: Results of the Experiment.

disappointing with respect to the generation and execution time. On the other hand, Configuration E and F are significantly more *efficient* on application Rachota, FreeMind, and JabRef, since fewer event sequences are generated and executed while preserving the same line and branch coverage.

Observation 5: For TerpWord, Configuration E attains a lower line and branch coverage than for Configuration D and F. Thus, the parameter *top* influences the quality of the selected event sequences. In TerpWord, there are a few events which have more than 50 dependent events. So, setting $top = 50$ might not be effective enough. The Configuration F achieves more coverage when setting $top = 100$.

Observation 6: For the grey-box Configurations D, E and F, increasing the length of the abstract event sequences does not significantly improve the coverage.

Observation 7: The number of *broken event sequences* is relatively low comparing to the total number of event sequences; they ranges between 0,5% and 1,4%. Broken event sequences are sequences sampled from the EFG, but could not be executed due to the limitations described in Section 4.2.

Observation 8: The experiment found 3 different bugs: The first bug is found in JabRef with Configuration B and Configuration D. A `NullPointerException` is thrown if the user clicks `Options`, `Manage custom imports`, `Add from folder`, `Cancel`. The bug is found in the black-box and in the grey-box approach using a sequence length of 2.

Observation 9: The second bug was found in JabRef with Configuration D. The following sequence of events causes an `ArrayOutOfBoundsException`: (1) In the main window, click `Manage content selectors`, which opens a new dialog; (2) switch to the main window and choose `Close database`. Then, (3) switch back to the previously opened dialog and click `OK`. The error occurs, because the new opened dialog is started *modeless*, which allows the user to close the database, although the dialog still suggests the user to modify the database.

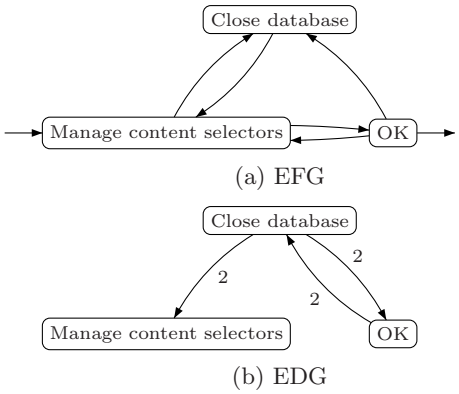
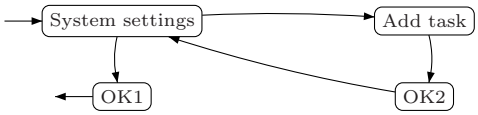


Figure 6: EFG and EDG snippet of JabRef.

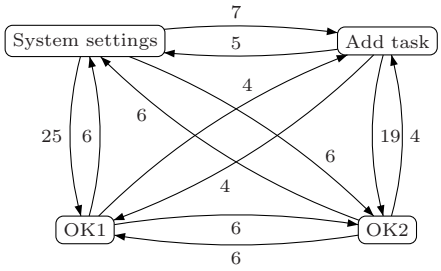
Figure 6 shows the EFG and EDG of JabRef that corresponds to the found bug. In the event sequence generation for event `Close database`, the grey-box approach detects the data dependency to event `OK`. This data dependency (weight = 2) consists of a field for JabRef's metadata, which is written in `OK` and read in `Close database`. Thus, the abstract event sequence ⟨`Close database`, `OK`⟩ is generated. This abstract event sequence is converted into an executable event sequence, because there exists no corresponding path in the EFG. Algorithm 3 picks the shortest path from an initial event to `Close database`, and the shortest path between succeeding events to `OK`, which leads to the following executable event sequence: ⟨`Manage content selectors`, `Close database`, `Manage content selectors`, `OK`⟩. The black-box approach will be able to detect this failure using a event sequence of length 4. However, it will first need to generate and execute all possible sequences of length 4.

Observation 10: The third bug was found in Rachota with Configuration D. The following sequence of events causes

a `NullPointerException` at restart: (1) Click on `System settings`; (2) Add a new task (`Add task`) and leave the text fields blank; (3) click the OK button (`OK2`). Then, (4) click on the OK button (`OK2`), that writes all tasks to a file. The errors occurs, because the new added task contains a `null` value when it is written to the user settings. Then, a null-reference is returned when the user settings are read, which is not correctly handled.



(a) EFG



(b) EDG

Figure 7: EFG and EDG snippet of Rachota.

Figure 7 shows the EFG and EDG of Rachota that corresponds to the bug. When generating abstract event sequences, we choose those sequences with the highest edge values first. For event `OK2`, the first abstract event sequence is ⟨`OK2`, `System settings`⟩, with a weight of 6. Our second abstract event sequence is ⟨`OK2`, `OK1`⟩, with a weight of 6. Since this abstract event sequence is not allowed to execute in its current form, it is converted into an executable event sequence using Algorithm 3. Hence, the final executable event sequence that can be run on the application is ⟨`System settings`, `Add task`, `OK2`, `System settings`, `OK1`⟩. The black-box approach will be able to detect this failure using an event sequence of length 5. However, the number of event sequences with length 5 would be 6,605,912, since Rachota consists of 154 events.

6. DISCUSSION

Regarding the research question of our experiment we found that the answer of **Q1** is **yes**: Considering possible data dependencies between events lead to fewer event sequences and decreases the time to run all test cases. The answer of **Q2** is **no**: We did not find enough evidences to show an improvement of the effectiveness.

The main result is that the grey-box approach in most cases produces a lot less test cases (each generated abstract event sequence represents one test case) than the black-box approach. Our initial assumption is that in GUIs several widgets carry out completely independent tasks. For example, a toolbox usually offers *save*, *print*, *copy*, *undo/redo* and *find*. However, *print* is unlikely to have a side effect on all other widgets. Thus, it is not efficient to test all combinations of *print* plus one other event (see Configuration B). Here, the grey-box approach can achieve significant improvements. For Rachota, more abstract event sequence are generated than in the black-box approach. The reason is that the events in Rachota have a lot of dependencies to other events. In this case, more edges in the EDG than in the EFG are obtained, and thus, more abstract event sequence are generated. However, we observe that in the other AUTs the number of EFG edges is higher than the number of EDG edges.

Increasing the length of event sequences in both approaches (black-box and grey-box) implies a considerably increase of the generation time. However, if we compare with the execution time, the generation time itself is not a big issue. Moreover, in practice, the testing process can be very limited in terms of resources and time to generate and execute all event sequences and test cases respectively. In this way, we could adapt our approach to an on-the-fly test case generation, where a specific timeout is given and parameter like *len* and *top* are not fixed, but vary in a range.

Using the grey-box approach, two different bugs were found, that were not found in the black-box approach, and, there are two main reasons: (1) all abstract event sequences incorporate data dependencies in the application's bytecode, and (2) the abstract event sequences have a non-fixed length. For instance, while Configuration A, B, and C select events that are directly connected, Configuration D, E, and F select events based on their data dependencies. Thus, the executable event sequence length in these grey-box configurations may vary comparing to the black-box configurations. Further, in the grey-box approach the length of an executable event sequence can be very long, e.g., if the distance of events in an abstract event sequence, in terms of intermediate events, is very high in the EFG.

The overall code coverage reported in our experiment is relatively low for several reasons. For instance, key strokes (`KeyListener`) and mouse gestures (`MouseListener`) are not yet considered, but frequently used in the application FreeMind, in order to draw a mind-map via mouse interactions. Support for these events in the GUI ripper and Replayer is scheduled for the future release of GUITAR. Moreover, the use of random input-data may lead to the execution default branches in the applications.

7. THREATS TO VALIDITY

We report 2 threats to internal validity. The first is the experiment replication. Almost all applications store user settings to the HDD, such as enabled and disabled toolbars, recently opened files etc. In order to ensure the precondition (i.e., the system's state) for each run of a test case, it is important that those user settings have to be deleted before execution. Otherwise, test cases may mistakenly fail, e.g., a GUI component is not found due to an existing user setting. In order to decrease this threat to internal validity we ran the experiment twice and got the same result.

The second is that some applications are strongly connected to the date and time of their execution. For instance, GUI components like *calendar controls* are considered in the GUI ripper and in the construction of the EFG. When replaying the test cases, some of them may fail, because the GUI components are not recognized anymore (during replaying the calendar control shows a different date as the calendar control was ripped).

One threat to external validity is the portability of the configurations. For instance, mobile phones have a different

environment and the construction of the EFG and EDG can be completely different. In principle, there is no reason to believe that the grey-box approach is not applicable to other platforms. To generalize the approach to other platforms, we must first port the ripper and replayer tools. Further, the model implementations have to be adapted to the corresponding environment. In this way, we believe that our approach can be generalized to different platforms.

8. RELATED WORK

Several approaches for modeling GUI-based applications have been developed for test case generation.

Model-based GUI testing: Different models can be used for event sequence generation [2, 25, 26]. For example, AI planning techniques are used in [22]; covering arrays in [28]. Event sequences are generated from these models and executed as test cases on the GUI to validate its behavior. In the grey-box approach, the EDG, created by analyzing bytecode, is used to generate event sequences. In [8], symbolic execution is used to find adequate inputs for event sequences. While symbolic execution is a powerful technique to find precise input values, it's applicability is limited due to the complexity of the used algorithms. In contrast, the grey-box approach only tries to identify simple data dependencies without tracking the actual value of fields, and thus it is applicable for reasonably sized applications. In [19], a method to dynamically observe a program's behavior at execution time is presented. Instead of analyzing the source code, an analysis of the call stack at run-time is performed. Event Sequences are then generated such that a minimum set covers a maximum possible set of program execution paths. In [17, 18] the AutoBlackTest approach is presented, which constructs a GUI model by learning how the GUI interacts with the system functionalities. Then, the tool selects an executable and non-redundant test suite. They also compare with the GUITAR approach. However, we could not empirically compare with AutoBlackTest since it is not available at the moment. In [29], feedback obtained by executing an event sequence is used to generate an improved test suite. It is an iterative method where GUI run-time feedback is used instead of source code information. In [13] the execution of a GUI-based application is represented as a sequence of events and output states. A state graph for the GUI is built which makes it possible to apply code based testing methods to GUIs.

Byte, Binary and Source Code Analysis: Many tools are available for reachability analysis and state space exploration of programs using the byte, binary, or source code. For example, JavaPathFinder [11] works at the Java bytecode level to identify deadlocks, assertion violations and other properties of the program using heuristics for reducing the state space explosion. Soot [15] is designed to be a framework to allow researchers to experiment with analyses and optimizations of Java bytecode. In the grey-box approach, we are interested in detecting sequences of events, which eventually bring the system to a failure state. Hence, we decide to implement a light-weight bytecode analysis, which can be enhanced by the support of alternative tools.

Search-based testing: In [1] a search-based testing technique is proposed. Unit tests for Java classes and methods are generated by looking for tests that satisfy given heuristics. Another approach using search-based testing is proposed in [6]. Heuristics are used to generate test cases that violate automated test oracles. In the grey-box approach, a data dependency can be seen as a heuristic, which helps to sample the user-level model (EFG) more efficiently.

The grey-box approach is similar to the generation of sequences of method calls, e.g., in libraries. However, when system testing an application through its GUI, not all methods (event handlers) may be available. For instance, a check box is likely to have no separate event handler, which changes the value from *selected* and *deselected*, once a user clicks this check box. This behavior may be implemented in the GUI framework and is not existing in the application itself. Without a user-level model it is difficult to generate a proper event sequence, if the value of the check box is evaluated in a further event handler (method) within the application, while it was changed in the GUI framework. Further, providing precise input for data-bound widgets, e.g., for an event handlers that governs a text box, is tough. Transferring input-data to a text box during test case execution, e.g., via reflection, may violate an invariant of the class. For instance, when the text box is disabled with regard to the event-flow, and does not accept any input.

9. CONCLUSION AND FUTURE WORK

We presented a new automatic grey-box approach for GUI event sequence generation. An EFG is generated automatically by observing the GUI at run-time (black-box). In addition, the application's bytecode is analyzed to find *data dependency* between event handlers (white-box) to generate a model called event-dependency graph (EDG). Abstract event sequences representing data dependencies are first generated from the EDG. These are then converted into Executable event sequences by looking up the EFG.

The grey-box approach incorporates 2 main steps: (i) model construction (EFG and EDG), and (ii) event sequence generation. The approach improves event sequence generation by producing fewer test cases and avoids generating event sequences where consecutive events share no data dependencies. Empirical evaluation shows that the grey-box approach decreases the time to generate event sequences and the time for executing test cases while retaining coverage.

Utilizing a black-box and a white-box model for the generation of event sequences is promising: We plan to improve the creation of the models and the generation of event sequences:

Model Creation: We plan to enhance the analysis of event handlers and the computation of the weight between two events respectively. Table 4 shows the potential for increasing the coverage of the AUT's. We believe that analyzing conditionals, i.e., if-, switch-, and loop-statements, can lead to the execution of more lines and branches. In the long run, the grey-box approach is supposed to provide a framework, where different black-box and white-box techniques can be plugged to generate event sequences. For instance, one would like to guide a dynamic symbolic execution [9] based on the EFG, or wrap a GUI application in a set of parameterized unit tests [7].

Event Sequence Generation: Typically, applications contain a subset of events with a relatively high number of dependent events, e.g., the system settings are read in many other event handlers. The grey-box approach enables us to identify these events, which we call *hot spots*. Intuitively, *hot spots* may be fault prone owing to inter-procedural data dependencies. In a future work, we plan to specifically analyze

hot spots while generating event sequences. More precisely, our event sequences generation uses the parameters *len* and *top* while generating a set of executable test cases. Considering *hot spots* might be useful to limit the event sequence generation and test case execution to a specific timeout. The idea is to spend a certain time on the testing of highly dependent events.

Evaluating the fault detection effectiveness is an important aspect of automated event sequence generation techniques. As a future work, we consider to evaluate the effectiveness of sequences generated from the EDG. We will start with fault-seeded versions of the application, and then naturally occurring faults in fielded applications. In order to have strong evidence about the experiment, we plan to execute the *best* configuration with different seeds.

Acknowledgments

The authors would like to thank Simon Pahl who supported us in the implementation. This work is partially supported by the research projects EVGUI and ARV funded by the Macau Science and Technology Development Fund, and the US National Science Foundation under grant CNS-0855055.

10. REFERENCES

[1] L. Baresi and M. Miraz. TestFul: automatic unit-test generation for Java classes. In *ICSE (2)*, pages 281–284, 2010.

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

[3] C. Bertolini, A. Mota, E. Aranha, and C. Ferraz. GUI Testing Techniques Evaluation by Designed Experiments. In *ICST*, pages 235–244, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[4] C. Bertolini, G. Peres, M. d'Amorim, and A. Mota. An Empirical Evaluation of Automated Black Box Testing Techniques for Crashing GUIs. In *ICST*, pages 21–30, 2009.

[5] R. C. Bryce, S. Sampath, and A. M. Memon. Developing a Single Model and Test Prioritization Strategies for Event-Driven Software. *IEEE Trans. Software Eng.*, 37(1):48–64, 2011.

[6] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE*, pages 416–419, 2011.

[7] G. Fraser and A. Zeller. Generating parameterized unit tests. In M. B. Dwyer and F. Tip, editors, *ISSTA*, pages 364–374. ACM, 2011.

[8] S. Ganov, C. Killmar, S. Khurshid, and D. Perry. Event Listener Analysis and Symbolic Execution for Testing GUI Applications. In *ICFEM*, pages 69–87, 2009.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[10] A. Grilo, A. Paiva, and J. Faria. Reverse Engineering of GUI Models for Testing. In *CISTI*, pages 1–6. IEEE Computer Society, 2010.

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

[12] A. Jääskeläinen, M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, T. Takala, and H. Virtanen. Automatic GUI test generation for smartphone applications - an evaluation. In *ICSE*, pages 112–122, 2009.

[13] M. R. Karam, S. M. Dascalu, and R. H. Hazimé. Challenges and opportunities for improving code-based testing of graphical user interfaces. *J. Comp. Methods in Sci. and Eng.*, 6:379–388, April 2006.

[14] N. Kicillof, W. Grieskamp, N. Tillmann, and V. Braberman. Achieving Both Model and Code Coverage with Automated Gray-box Testing. In *A-MOST*, pages 1–11. ACM, 2007.

[15] J. Lhoták, O. Lhoták, and L. J. Hendren. Integrating the Soot Compiler Infrastructure Into an IDE. In *CC*, pages 281–297, 2004.

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

[17] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. AutoBlackTest: a Tool for Automatic Black-box Testing. In *ICSE*, pages 1013–1015, 2011.

[18] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In *ICST*, Montreal, Canada, 2012. IEEE Computer Society.

[19] S. McMaster and A. M. Memon. Call-Stack Coverage for GUI Test Suite Reduction. *IEEE Trans. Software Eng.*, 34(1):99–115, 2008.

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

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

[22] A. M. Memon, M. E. Pollack, and M. L. Soffa. Plan Generation for GUI Testing. *AIPS*, pages 226–235, 2000.

[23] A. M. Memon and Q. Xie. Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software. *IEEE Trans. Softw. Eng.*, 31:884–896, October 2005.

[24] M. J. Osborne and A. Rubinstein. *A course in game theory*. The MIT Press, July 1994.

[25] A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal. Modeling and Testing Hierarchical GUIs. In *Abstract State Machines*, pages 329–344, 2005.

[26] H. Reza, S. Endapally, and E. S. Grant. A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets. *ITNG*, pages 366–370, 2007.

[27] 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*, pages 181–186, New York, NY, USA, 2010. ACM.

[28] X. Yuan, M. B. Cohen, and A. M. Memon. GUI Interaction Testing: Incorporating Event Context. *IEEE Trans. Software Eng.*, 37(4):559–574, 2011.

[29] X. Yuan and A. M. Memon. Iterative execution-feedback model-directed GUI testing. *Information & Software Technology*, 52(5):559–575, 2010.