Lightweight Static Analysis for GUI Testing

Stephan Arlt*, Andreas Podelski*, Cristiano Bertolini†, Martin Schäf‡, Ishan Banerjee§, Atif M. Memon§

*Albert-Ludwigs-Universität Freiburg
†Federal University of Pernambuco
‡United Nations University Macau
§University of Maryland

***Abstract*—GUI testing is an active research area. The open challenge is the judicious generation of event sequences (an event sequence encodes a user interaction). A major advance in this direction is the use of a black-box model to systematically generate event sequences that are executable on the GUI. The black-box model can be, e.g., an Event Flow Graph (EFG) or an Event Sequence Graph (ESG). In this paper we propose a new approach to select relevant event sequences among the event sequences generated by a black-box model. We express the relevance of an event sequence by a precisely defined dependency between a fixed number of events in the event sequence. Departing from a pure black-box approach we apply a static analysis to the bytecode of the application. This allows us to infer a dependency graph, which we call Event Dependency Graph (EDG). We use the EDG together with a black-box model to construct a set of relevant event sequences among the executable ones. We have implemented our approach in a new tool. We evaluate the approach on four open source GUI applications. With the specific choice of a lightweight static analysis, the approach scales to large applications and, at the same time, leads to an informed selection of event sequences. Using our approach we are able to find previously undetected bugs.**

***Keywords*- GUI Testing; Test Automation; Black-box Testing; Static Analysis;**

I. INTRODUCTION

GUI testing is an active research area; see [2], [3], [4], [5], [9], [12], [18], [19]. Testing a software application through its graphical user interface (which is a form of system testing) is difficult because the space of possible user interactions is unrestricted. A user action, such as clicking a mouse button, triggers an *event* in the application. An application responds to an event by executing a piece of code, the *event handler* for that event. Therefore, an *event sequence* is an integral part of a GUI test case. Given the unrestricted space of possible event sequences, the open challenge is the judicious generation of event sequences. The goal is to select a reasonably sized and effective set of event sequences.

A major advance towards this goal is the use of a black-box model to systematically generate event sequences; see [2], [3], [12], [18], [19], [21]. Here, the black-box model is a graph whose nodes are events and whose paths represent event sequences; it is called EFG (Event Flow Graph) in [12] and ESG (Event Sequence Graph) in [2]. The idea is that an event sequence on a path can be used for a GUI test case.

Another interesting direction is the development of a white-box approach for GUI testing; see [6], [15], [16]. For example, in [6] symbolic execution is used to generate input data for GUI test cases. Until now, however, a white-box approach was not used to generate event sequences for GUI test cases. The idea would be to use an analysis of the source code for an informed selection of event sequences. It is open, however, whether an analysis of the source code can realistically infer which event sequences are executable on the GUI.

The question is whether there is an approach to GUI testing which inherits from the best of the two worlds: the applicability to realistic GUI applications from a black-box approach and the informedness to select test cases from a white-box approach. Put into a slogan, the black-box tells us what we *can* test, the white-box tells us what we *should* test. Phrasing the slogan negatively, the black-box may produce *irrelevant* test cases; the white-box may produce *non-executable* test cases.

In this paper we propose a new approach to select relevant event sequences among the event sequences generated by a black-box model. We express the relevance of an event sequence by a precisely defined dependency between a fixed number of events in the event sequence. In a step that starts from a pure black-box approach and departs from it, we extract the set of events of the GUI and the corresponding event handlers. In a step typical for a white-box approach, we apply a static analysis to the bytecode of the GUI application and its dependent libraries. This allows us to infer a dependency relation between events. The relation gives rise to a graph, which we call Event Dependency Graph (EDG). The EDG can be used to infer the relevance of an event sequence; it says nothing about its executability. In contrast, a black-box model can be used to infer the executability of an event sequence. In a step that goes back towards the black-box approach, we use the EDG together with a black-box model (here, the EFG) to construct an informed selection of executable test cases, namely through the set of relevant event sequences among the executable ones.

We have implemented our approach in a new tool called *Gazoo*. We evaluate the approach on four open source GUI applications. With a specific choice of a lightweight static analysis, the approach scales to large applications and, at the same time, leads to an informed selection of event sequences. Using our approach we are able to find previously undetected bugs.

Structure of the paper. In the next section we illustrate the application of our approach using an example of a GUI application. Then we present the approach and its implementation. In the Experiments section we formulate the research questions and evaluate the approach.

II. Example

In this section, we illustrate the application of our approach on an example of a GUI application; see Figure 1. The example contains an oversimplified version of a bug which we detected in an existing GUI application; see Section IV. Note that the GUI application must not be confused with the GUI toolkit: We here concentrate on testing the event handlers of the GUI application instead of testing certain features of the GUI toolkit.

Figure 1. Example GUI. The arrows between the two screenshots of the GUI indicate the transition between two views. Clicking the button for the event e_3 leads to opening the `Dialog` window (and hiding the `MainWindow`; i.e., the buttons for the events e_1, e_2, and e_3 are no longer enabled). Clicking the button for the event e_4 closes the `Dialog` window and leads back to the state in the first view; i.e., the buttons for the events e_1, e_2, and e_3 are enabled again.

The description of the possible user interactions with the GUI in Figure 1 reveals its *event flow*. Namely, each event can follow any other event except that e_4 can follow only after e_3. The event flow (i.e., the can-follow relation between events) is expressed by the graph depicted in Figure 2. Such a graph, called EFG (Event Flow Graph), represents the black-box model that is used to generate test cases in [12]. The idea is that a path in the EFG encodes a possible user interaction. The marking of e_1, e_2, and e_3 as initial events encodes how a user interaction can start. An EFG can be constructed automatically using reverse engineering [13].

We will first explain how a *pure black-box approach* à la [12] would work on the example GUI. It would first construct the EFG in Figure 2. It would then use the EFG to generate the ten *event flow sequences* in Figure 3, which it would transform into the ten *test sequences* in Figure 4 (and embed those into test cases).

An *event flow sequence* consists of the events on one of the paths in the EFG. The length of the paths in the EFG is fixed as the *parameter* of the approach; here we set the parameter to $n = 2$.

The transformation of an event flow sequence into a *test sequence* accounts for the fact that a user interaction must start in an initial event, namely by a suitable expansion. Specifically, since the event e_4 is not an initial event in the EFG, a test sequence cannot start with e_4. Thus, the transformation expands s_8, s_9, and s_{10} by an initial event, here e_3. The transformation does not expand s_1, …, s_7 since they already start with an initial event.

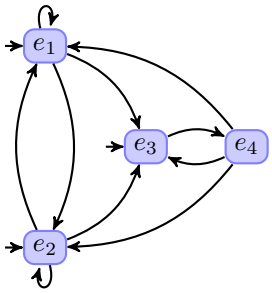


Figure 2. Event Flow Graph (EFG) for the example GUI. Each event can follow any other event except that e_4 can follow only after e_3. A path in the EFG encodes a possible user interaction. The marking of initial events encodes that a user interaction can start with e_1, e_2, e_3, but not e_4.

$$
\begin{array}{ll}
s_1 = \langle e_1, e_1 \rangle & s_6 = \langle e_2, e_3 \rangle \\
s_2 = \langle e_1, e_2 \rangle & s_7 = \langle e_3, e_4 \rangle \\
s_3 = \langle e_1, e_3 \rangle & s_8 = \langle e_4, e_1 \rangle \\
s_4 = \langle e_2, e_1 \rangle & s_9 = \langle e_4, e_2 \rangle \\
s_5 = \langle e_2, e_2 \rangle & s_{10} = \langle e_4, e_3 \rangle
\end{array}
$$

Figure 3. The event flow sequences extracted from the EFG in Figure 2 (i.e., the sequences of events on a path in the EFG of length $n = 2$).

$$
\begin{array}{ll}
t_1 = \langle e_1, e_1 \rangle & t_6 = \langle e_2, e_3 \rangle \\
t_2 = \langle e_1, e_2 \rangle & t_7 = \langle e_3, e_4 \rangle \\
t_3 = \langle e_1, e_3 \rangle & t_8 = \langle e_3, e_4, e_1 \rangle \\
t_4 = \langle e_2, e_1 \rangle & t_9 = \langle e_3, e_4, e_2 \rangle \\
t_5 = \langle e_2, e_2 \rangle & t_{10} = \langle e_3, e_4, e_3 \rangle
\end{array}
$$

Figure 4. The test sequences that would be generated in the pure black-box approach whose parameter is $n = 2$. They are obtained from the event flow sequences in Figure 3, possibly by expansion (a test sequence cannot start with e_4).

Our approach. We will now explain how our approach works on the example GUI; see Figure 9.

In the first step, we extract the event handlers of the events e_1, …, e_4 from the GUI application and its bytecode. The Java source code is shown in Figure 5. For the purpose of this example, we use the source code (and not, as in real, the bytecode).

In the second step of our approach, we apply a static analysis to the bytecode and derive a *dependency* relation between the four events of the GUI: the event e_1 writes the field `text` which is read in the event e_4; so does the event e_2, and so does the event e_4 itself. The dependency relation between events ("writes-to/reads-from") is expressed by the graph depicted in Figure 6. We call it the Event Dependency Graph (EDG). In this paper we use the terminology of [23] for the write/read dependency.

In the third step, we first generate the *event dependency sequences* shown in Figure 7 from the EDG in Figure 6. We then transform the event dependency sequences into

```
1  class MainWindow extends JFrame {
2    String text = new String();
3
4    // handler for event e1
5    void e1() {
6      text = "Hello World";
7    }
8
9    // handler for event e2
10   void e2() {
11     text = null;
12   }
13
14   // handler for event e3
15   void e3() {
16     Dialog d = new Dialog(this);
17     d.setVisible(true);
18   }
19
20   class Dialog extends JDialog {
21     // handler for event e4
22     void e4() {
23       text = text.trim();
24       Dialog.this.setVisible(false);
25     }
26   }
27 }
```

Figure 5. The event handlers extracted from the example GUI and the bytecode. The class `MainWindow` defines the event handlers for e_1, e_2, and e_3, and the class `Dialog` for e_4. The event handler e_1 assigns a string value to the field `text` (line 6). The event handler e_2 sets the field `text` to `null` (line 11). The event handler e_3 opens the dialog (line 16-17). The event handler e_4 assigns the trimmed string value of the field `text` to the field `text`, and closes the dialog (line 23-24).

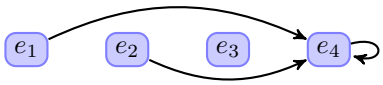


Figure 6. Event Dependency Graph (EDG) for the example GUI. Each edge expresses a write/read dependency found by the static analysis (applied to the source code shown in Figure 5): the event e_1 writes the field `text` which is read in the event e_4; so does the event e_2, and so does the event e_4 itself. We observe that a path in the EDG may not be matched by any user interaction (since e_4 can follow only after e_3).

the test sequences shown in Figure 8, hereby using the EFG in Figure 2 in an auxiliary procedure.

An *event dependency sequence* consists of the events on a path in the EDG. Now, it is the length of the paths in the EDG which is fixed by the parameter of our approach; here we set the parameter to $n = 2$ (i.e., the same value as in the pure black-box approach).

$$s_1 = \langle e_1, e_4 \rangle \quad s_3 = \langle e_4, e_4 \rangle$$
$$s_2 = \langle e_2, e_4 \rangle$$

Figure 7. The event dependency sequences extracted from the EDG in Figure 6 (i.e., the sequences of events on a path in the EDG of length $n = 2$).

$$t_1 = \langle e_1, e_3, e_4 \rangle \quad t_3 = \langle e_3, e_4, e_3, e_4 \rangle$$
$$t_2 = \langle e_2, e_3, e_4 \rangle$$

Figure 8. The test sequences that are generated in our approach when the parameter is set to $n = 2$. They are obtained from the event dependency sequences in Figure 7, using the EFG in Figure 2; the event e_4 can follow only after e_3 (and a test sequence cannot start with e_4).

The transformation of an event dependency sequence s into a test sequence t consists of expanding the sequence until it corresponds to a path in the EFG and then further until it corresponds to an EFG path that starts in an initial event. For example, we transform s_1 into t_1 by inserting e_3 between e_1 and e_4. In the case of the transformation from s_3 into t_3, we need to additionally add an initial event (since a test sequence cannot start in e_4). Thus, the transformation proceeds in two steps where the second step is analogous to the transformation from the event flow sequences in Figure 3 into the test sequences in Figure 4. I.e., in the pure black-box approach one would take all event flow sequences of the length fixed by the parameter $n = 2$. In our approach, we take a few (three) selected event flow sequences whose length is *a priori* unbounded.

Our approach finds a bug in the example. The execution of one of the three test sequences generated in our approach, namely t_2, causes a `NullPointerException`. This is because the field `text` is set to `null` in the event handler e_2 and then de-referenced in event handler e_4. The intermediate event e_3 in the test sequence t_2 is not involved in the causal chain of the bug; it is needed because without it, the event sequence would not be executable, i.e., could not be executed on the GUI application.

Would the pure black-box approach find the bug?

No. In our example, the set of the three test sequences generated from event dependency sequences of length $n = 2$ is disjoint from the set of the ten test sequences generated from event flow sequences of length $n = 2$. Thus, the test sequence t_2 cannot be generated from one of the ten event flow sequences of length $n = 2$. I.e., the bug cannot be detected in the pure black-box approach when the parameter is set to $n = 2$.

Yes. In our example, the set of the three test sequences generated from event dependency sequences of length $n = 2$ is a subset of the (24) test sequences generated from event dependency sequences of length $n = 3$. Thus, the test sequence t_2 could be generated from one of the event flow sequences of length $n = 3$. I.e., the bug could be detected in the pure black-box approach when the parameter was set to $n = 3$, at least in this example. However, setting $n = 3$ can be considerably expensive in terms of execution time/number of test sequences.

No. We can modify the example such that the bug can no longer be detected in the pure black-box approach even when the parameter is set to $n = 3$ (the modification extends to $n = 4$, $n = 5$, ...). Namely, we add an intermediate dialog box with the event e_4^1 such that the test sequence t_2 becomes non-executable and the bug would be detected only by the test sequence t_2' below.

$$t_2' = \langle e_2, e_3, e_4^1, e_4 \rangle$$

If the button for e_4^1 can only be activated by e_3 and the button for e_4 can only be activated by e_4^1, then the edge (e_3, e_4) in the EFG of Figure 2 gets replaced by the two edges (e_3, e_4^1) and (e_4^1, e_4). This means that the test

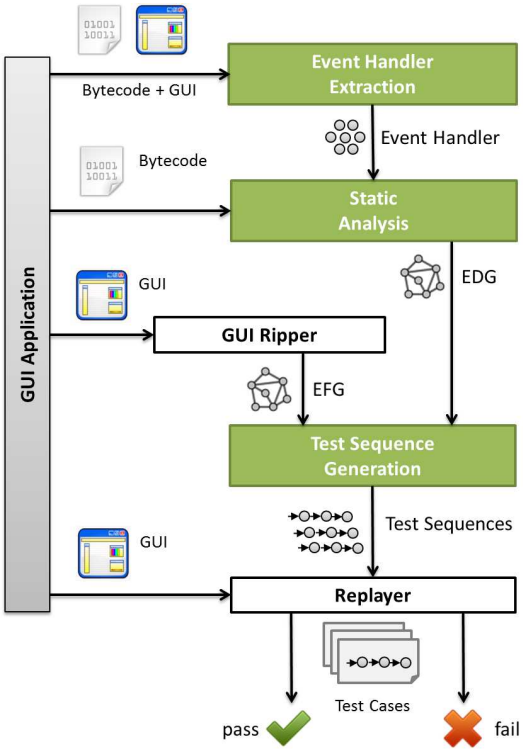


Figure 9. Our approach, with three main steps. (1) The step *Event Handler Extraction* extracts a set of event handlers from the bytecode and the GUI of an application. (2) The step *Static Analysis* constructs an EDG from the write/read dependencies which are found by a static analysis applied to the event handlers and the bytecode. (3) The step *Test Sequence Generation* generates a set of test sequences from the EDG, with the help of the EFG. There are two more steps: the step GUI Ripper constructs an EFG, and the step Replayer executes the generated test sequences on the GUI (reporting sets of *passed* and *failed* test cases).

sequence t'_2 can no longer be generated from an event flow sequence of length $n = 3$. I.e., the bug cannot be detected in the pure black-box approach even when the parameter is set to $n = 3$.

One might argue that one can modify the example such the bug can no longer be detected in our approach with the parameter $n = 2$. This, however, would involve constructing a more complex bug, namely one whose causal chain involves more than two events.

III. Approach and Implementation

Our approach consists of three main steps as shown in Figure 9: (1) the extraction of event handlers; (2) the static analysis; and (3) the generation of test sequences.

A. Event Handler Extraction

Each event in a GUI (e.g., a click on a button `OK`) is encoded as an *event handler* (e.g., *OnClickOK*). The step Event Handler Extraction mixes aspects of black-box and white-box approaches in order to extract the GUI's event handlers. First, as in a black-box approach, we execute the GUI application and enumerate the GUI's widgets (e.g., windows, buttons, and text fields). Then, leaving a black-box approach, we apply Reflection[1] to obtain the Java object corresponding to each widget (e.g., a `JButton` object). We ask the Java object to invoke the method `getActionListeners`. The method invocation returns the widget's event handlers (which are called *action listeners* in Java). The widget's ID is used as a unique identifier for each event. If the Java object does not provide the method `getActionListeners`, then there exists no registered event handler to this widget. In this case, we can simply discard the corresponding event for the construction of the EDG; it will still be used for the construction of the EFG.

B. Static Analysis

The input of the step Static Analysis is the bytecode of the application and the set of event handlers which has been extracted by the step Event Handler Extraction. For each event handler, we infer *a sound approximation* of what fields can be written to resp. read from during the execution of the event handler. The inference is done by a lightweight static analysis applied to the bytecode. Concretely, for each event handler, the static analysis collects all fields that are written to resp. read by the event handler directly or indirectly (indirectly means: by all methods that it may possibly call, directly or via intermediate method calls). The static analysis returns two mappings that assign to each event e a set of fields, namely $FieldsWritten(e)$ resp. $FieldsRead(e)$. We construct the EDG from the two mappings. Namely, the edge (e, e') belongs to the EDG if the intersection between $FieldsWritten(e)$ and $FieldsRead(e')$ is not empty.

The EDG is the output of the step Static Analysis. An edge (e, e') in the EDG indicates a *dependency* between the events e and e', meaning: the event handler of e' possibly reads data written by the event handler of e.

Implementation details. We use the ASM framework[2] for the implementation of the static analysis described above. We apply the static analysis to the bytecode, and not on the Java source code as shown for the example discussed in Section II. For concreteness, we show the bytecode for the event handler e_2 and e_4 in the example; see Figure 10. Our earlier implementation efforts, which used a static analysis on the source code (implemented in JDT[3]), failed; we did not succeed to obtain the required information. For each event handler, the static analysis collects all fields written by analyzing the instruction `PUTFIELD` [10], and all fields read by analyzing the instruction `GETFIELD`. Since an event handler may directly or indirectly call methods, the static analysis follows all method calls by analyzing the instruction `INVOKE`.

C. Test Sequence Generation

The input to the step Test Sequence Generation is the EDG, which has been constructed in the step Static Analysis, and the EFG, which has been constructed in an auxiliary step which we will describe later. As we have described above, the EDG encodes the dependency

[1] http://java.sun.com/developer/technicalArticles/ALT/Reflection/

[2] http://asm.ow2.org/

[3] http://www.eclipse.org/jdt/

```
1 void e2()V
2  ICONST_0
3  PUTFIELD MainWindow.text : Z
4
5 void e4()V
6  GETFIELD MainWindow.text : Ljava/lang/String;
7  INVOKEVIRTUAL java/lang/String.trim()Ljava/
       lang/String;
8  PUTFIELD MainWindow.text : Ljava/lang/String;
```

Figure 10. Bytecode for the event handlers for e_2 and e_4 in the example of Section II. The static analysis adds the `text` field of `MainWindow` to the set of fields written by e_2, and it adds it to the set of fields read by e_4. It thus infers the write/read dependency from e_2 to e_4. Thus, we add the edge (e_2, e_4) to the EDG. In bytecode, fields are read by the instruction `GETFIELD`; they are written by `PUTFIELD`. Methods are called using the `INVOKE` instruction (`INVOKEVIRTUAL`, `INVOKESTATIC`, `INVOKESPECIAL`). Line 2: the constant value 0 is pushed to the stack. Line 3: the constant value 0 is assigned to the field `text`. Line 6: the field `text` is read and pushed to the stack. Line 7: the method `trim` is called and the return value is pushed to the stack. Line 8: the return value is pop-ed from the stack and written to the field `text`.

relation between events, and the EFG encodes the flow relation between events. A path in the EDG, which we call *event dependency sequence*, is in general not executable, i.e., does not encode a user interaction that can be executed on the GUI. In contrast, a path in the EFG that starts in an initial event, which we call *test sequence*, is executable. Therefore, in order to obtain a test sequence from an event dependency sequence, we need to transform it until it forms a path in the EFG that starts with an initial event.

The length of the event dependency sequences is fixed by the parameter n of the overall approach. It is the step Test Sequence Generation, and only this step in the overall approach, that depends on the parameter n. In the example of Section II and in the experiments which we will discuss in Section IV, the parameter is set to $n = 2$.

The step Test Sequence Generation consists of two substeps: (1) the extraction of all event dependency sequences from the EDG, and (2) the transformation of event dependency sequences to test sequences. The transformation consists of finding a path in the EFG that contains all events of the given event dependency sequence, in the same order but not necessarily consecutive.

For two events in an event dependency sequence, say e and e', we compute the shortest path in the EFG from e to an initial event, and the shortest path from e to e'. We then concatenate both computed paths (i.e., the prefix of e and the suffix starting in e) in order to obtain an executable test sequence. In particular, we apply a breadth-first search on the EFG to find the shortest path between events. The search for the shortest path is motivated by the idea of having compact test cases (in terms of numbers of events in a test sequence).

There are two cases when an event dependency sequence cannot be transformed into a test sequence: (a) there exists no path in the EFG from an initial event to the first event of the event dependency sequence (we then discard the event dependency sequence); and (b) for two events in the event dependency sequence, say e and e', there exists no path in the EFG leading from e to e'. In this case, we first split the event dependency sequence into two test sequences, i.e., we compute the shortest path to an initial event for e and for e'. Then, we concatenate these two test sequences with a restart of the GUI application in between.

The need of a restart arises from the following observation: GUI applications tend to store *user settings*, e.g., the recently opened files in the File menu. For example, an event e writes a user setting which is read by an event e', but there exists no event flow between these two events. In order to test (e, e'), we generate two test sequences and concatenate them with a restart in between.

Auxiliary Steps

The step GUI Ripper constructs an EFG by executing the GUI application [13]. For completeness we describe it briefly; see [13] for details. The execution is directed to explore the hierarchical structure of the GUI in a depth-first manner. For each widget found during the execution, say a button `OK`, the GUI ripper triggers the assigned event, i.e., a button click. By recording the history of triggered events, the GUI Ripper detects the event flow (i.e., which pairs of events can be consecutive) and stores it in the EFG. The GUI Ripper uses a widget ID as a unique identifier for each event, just as the step Event Handler Extraction. In our implementation, the steps Event Handler Extraction and GUI Ripper are combined.

The step Replayer takes as input a set of test sequences and embeds each test sequence into a test case. A GUI test case consists of four components: (1) a *precondition* that must hold before executing a test sequence; (2) the *test sequence* to be executed; (3) possible *input data* to the GUI; and (4) a *postcondition* that must hold after executing the test sequence. The step Replayer executes all test cases obtained from the test sequences. That is, it ensures the precondition, executes the test sequence on the GUI application, inserts input data where necessary, and checks if the postcondition holds. If it holds, the step Replayer reports the test case as passed, and if not, as failed.

We have implemented our approach in a new tool called *Gazoo*. Our implementation uses an adaptation of the GUI Ripper and the Replayer of GUITAR[4]. In the next section we report the experiments conducted with the new tool.

IV. Experiments

In this section we evaluate our approach by a comparison with the pure black-box approach in [12]. We first present the setup of the experiments. Then we discuss the results of the experiments. We define the following five research questions:

- **Q1**: Does our approach scale to realistic GUI applications? A priori, this is not clear. The question is critical for a particular reason. In the pure black-box approach, one can bound the cost by fixing the parameter to, say, $n = 2$ (the approach will then generate at most k^2 test sequences where k is the

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

number of events). In our approach, the parameter fixes the complexity of the bug to be found (the approach guarantees to find all bugs whose causal chain involves $n = 2$ events); a priori, there is no bound on the number of the generated test sequences (nor on their length).

- **Q2**: Is the test sequence generation in our approach effectively selective; i.e., does it discard an interesting number of *irrelevant* event flow sequences?
- **Q3**: How much would we have to increase the parameter n for the pure black-box approach in order to generate all the test sequences that are generated by our approach with the parameter set to $n = 2$?
- **Q4**: Does our approach achieve the same coverage even in the cases when the number of generated test sequences is smaller than in the pure black-box approach?
- **Q5**: Is our approach effective for finding bugs?

A. Setup of the Experiments

We evaluate our approach using four Java-based open source applications: *JabRef 2.7* manages bibliographic references, *FreeMind 0.9* creates mind maps, *TerpWord 4.0* is a word processor, and *Rachota 2.3* is a time recording system. For Rachota, we used the artifacts from *Community Event-based Testing (COMET)*[5]. We choose these applications to consider both small and large applications (in terms of # of classes), and to cover different code styles. Figure 11 shows some relevant statistics of the AUTs (Applications Under Test).

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

Figure 11. Comparison between AUTs: number of lines of code (LOC), classes, events, and edges in the EDG and the EFG that are computed in the experiments. For Rachota, the size of the EDG is almost 50% higher than the size of the EFG.

Our experiments consist of two configurations. The configuration **EFG-2** stands for the pure black-box approach [12] where the parameter is set to $n = 2$; this limits the length of the considered paths in the EFG to $n = 2$. The configuration **EDG-2** stands for our approach where the parameter is set to $n = 2$ as well; this limits the length of the considered paths in the EDG to $n = 2$. The choice of the parameter $n = 2$ is motivated by previous empirical studies on bugs in GUI applications [20].

As mentioned in Section III, each test sequence is embedded into one test case consisting of (1) a precondition; (2) a test sequence; (3) input data; (4) a postcondition. For (1), as a precondition we define that all user settings of an AUT have to be deleted before executing the test sequence. For (3), we generate random data, i.e., random strings for text boxes. The computation of suitable input data (see [6], [7], [17]) for widgets represents an orthogonal problem and is not in the scope of this paper. For (4), we use a crash monitor as an oracle; this is a simple but reasonable oracle. In particular, we record any exception occurred during test case execution, and we automatically observe whether a test case is executable on a GUI. For a discussion of alternative oracles we refer to [14].

The test cases are executed on 10 virtual Linux machines with 2.0 GHz CPU, 2 GB RAM, 500 GB HDD. In order to mitigate the effect of randomness, the configurations EFG-2 and EDG-2 are executed three times. The total number of executed test cases amounts to 236,808.

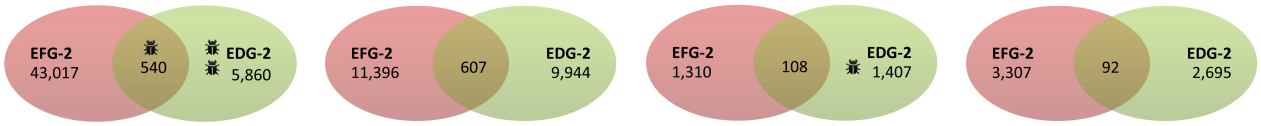
B. Results of the Experiments

Figures 12, 13, 14 and 15 and part of Figure 11 show the results of the experiments.

AUT	EFG-2	EDG-2
JabRef		
# test sequences	**43,017**	**5,860**
generation time (m)	93	21
generation time per test sequence (s)	0.13	0.22
execution time (h)	358	49
line coverage (%)	54	54
branch coverage (%)	26	26
# detected bugs	🐞	🐞🐞🐞
FreeMind		
# test sequences	**11,396**	**9,944**
generation time (m)	25	34
generation time per test sequence (s)	0.13	0.21
execution time (h)	98	88
line coverage (%)	53	53
branch coverage (%)	37	37
# detected bugs	-	-
Rachota		
# test sequences	**1,310**	**1,407**
generation time (m)	3	4
generation time per test sequence (s)	0.14	0.17
execution time (h)	6	6
line coverage (%)	61	62
branch coverage (%)	34	36
# detected bugs	-	🐞
TerpWord		
# test sequences	**3,307**	**2,695**
generation time (m)	7	8
generation time per test sequence (s)	0.13	0.18
execution time (h)	12	10
line coverage (%)	55	55
branch coverage (%)	36	36
# detected bugs	-	-

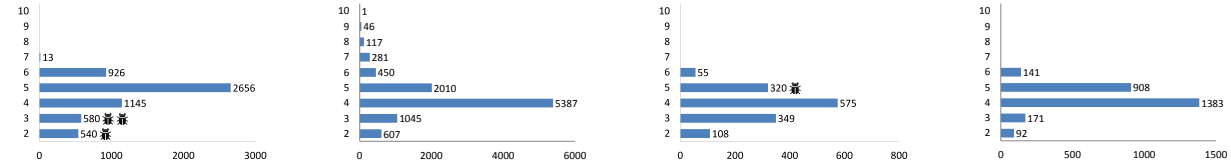
Figure 12. The configuration **EFG-2** stands for the pure black-box approach [12] where the parameter is set to $n = 2$; this limits the length of the considered paths in the EFG to $n = 2$. The configuration **EDG-2** stands for our approach where the parameter is set to $n = 2$ as well; this limits the length of the considered paths in the EDG to $n = 2$.

[5]http://comet.unl.edu/



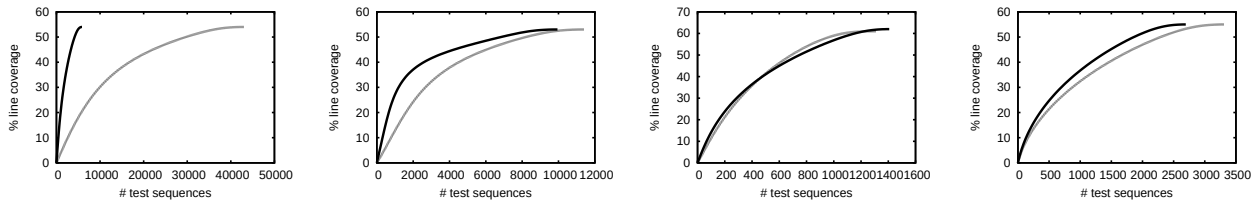
(a) JabRef (b) FreeMind (c) Rachota (d) TerpWord

Figure 13. In each Venn diagram, the set marked EFG-2 consists of the test sequences generated with configuration EFG-2, i.e., by the pure black-box approach. The set marked EDG-2 consists of the test sequences generated with configuration EDG-2, i.e., by our approach. Among the test sequences in EFG-2 our approach discards all those that are not in the intersection (they are not relevant test sequences).



(a) JabRef (b) FreeMind (c) Rachota (d) TerpWord

Figure 14. The distribution of the test sequences obtained from the event dependency sequences of length 2. The y-axis stands for the parameter which is required for the pure black-box approach in order to generate the test sequence. For example, 2,656 test sequences out of the (5,860) test sequences generated by our approach with parameter $n = 2$ are generated by the pure black-box approach only if the parameter is increased to $n = 5$.



(a) JabRef (b) FreeMind (c) Rachota (d) TerpWord

Figure 15. The trend of the achieved coverage. The x-axis indicates the number of executed test sequences. The y-axis indicates the line coverage. The black line represents the coverage of EDG-2 test sequences; the grey line represents the coverage of EFG-2 test sequences. We observe that the configuration EDG-2 does not loose coverage. Moreover, the configuration EDG-2 can achieve coverage of EFG-2 significantly faster.

Regarding research question **Q1**, we refer to Figure 12. Contrary to our initial worries, the number of test sequences does not explode. In the middle-sized application Rachota, it is slightly larger. In the somewhat larger application JabRef, it is significantly smaller. A potential explanation is that while our approach selects a test sequence for *every* pair of events with a write/read dependency, it selects *only* one. Another major worry of ours (based on earlier experience) concerned the scalability of the static analysis which is an essential ingredient of our approach. It seems that with our choice of a lightweight static analysis, we have identified a sweetspot in the precision/cost tradeoffs. Looking at the generation time and the generation time per test sequence, we conclude that the overhead incurred by the static analysis is acceptable. In summary, the answer to the research question **Q1** is **Yes**: our approach does scale to realistic GUI applications.

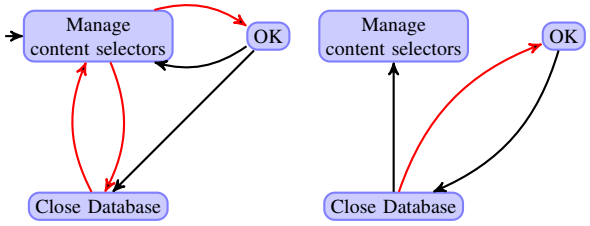
Regarding research question **Q2**, we refer to Figure 13. In each Venn diagram, the set marked EFG-2 consists of the test sequences generated with configuration EFG-2, i.e., by the pure black-box approach. The set marked EDG-2 consists of the test sequences generated with configuration EDG-2, i.e., by our approach. Among the test sequences in EFG-2 our approach discards all those that are not in the intersection. Among all test sequences in EFG-2, only the ones that are also in EDG-2 (i.e., in the intersection) are justifiably relevant (because they are known to contain a pair of two events with a write/read dependency). All other test sequences in EFG-2 are irrelevant, i.e., their selection is not based on the formal criterion of relevance and, hence, they are discarded by our approach. In JabRef only 540 out of the 43,017 event flow sequences in EFG-2 are relevant, i.e., 99% of the event flow sequences in EFG-2 are irrelevant and discarded by our approach. For the other AUTs, the numbers are similar: 95% of the event flow sequences in FreeMind; 92% for Rachota; 97% for TerpWord. In summary, the answer to the research question **Q2** is **Yes**: our approach discards an interesting number of irrelevant event flow sequences.

Regarding research question **Q3**, we refer to Figure 14. The y-axis stands for the parameter which is required for the pure black-box approach in order to generate the test sequence. For example, 2,656 test sequences out of the (5,860) test sequences generated by our approach with parameter $n = 2$ are generated by the pure black-box approach only if the parameter is increased to $n = 5$. A

similar fact holds true for each of the applications: for a rather large fraction of the test sequences generated by our approach, one has to increase the parameter to $n = 4$ or $n = 5$. To answer the research question **Q3**, one has to increase the parameter to $n = 7$ (for JabRef), to $n = 10$ (for FreeMind), to $n = 6$ (for Rachota and TerpWord) for the pure black-box approach in order to generate all the test sequences that are generated by our approach with the parameter set to $n = 2$.

Regarding research question **Q4**, we refer to Figure 15. The x-axis indicates the number of executed test sequences. The y-axis indicates the line coverage. The black line represents the coverage of the EDG-2 test sequences, the grey line represents the coverage of the EFG-2 test sequences. Since our static analysis recognizes event dependencies in depending libraries, we consider coverage as the sum of the coverage of the GUI application itself and its depending libraries. In order to plot a *fair* coverage trend (the order of the executed test sequences matters) we followed this procedure: First, we executed each test sequence and measured its achieved coverage. Second, we put the coverage results in a so-called *coverage sequence*. Third, we randomly modified the order in the coverage sequence k times using different seeds, where k is the number of all test sequences. Furthermore, we calculated the coverage trend of each modified coverage sequence. Fourth, in Figure 15 we reported the average of all coverage trends. We observe that the configuration EDG-2 does not loose coverage comparing to the configuration EFG-2. Moreover, we observe that EDG-2 can achieve the same coverage of EFG-2 significantly faster (in terms of the number of executed test sequences). For JabRef, the EDG-2 needs 5,860 test sequences to achieve the coverage of EFG-2, which needs 43,017 test sequences. Rachota makes an exception: EDG-2 achieves a slightly higher coverage with a slightly higher number of test sequences than EFG-2. Note that the EDG is larger than the EFG; see Figure 11. In summary, the answer to the research question **Q4** is **Yes**: our approach does achieve the same coverage even in the cases when the number of generated test sequences is smaller than in the pure black-box approach.

The answer to the research question **Q5** is **Yes**. Our approach is able to find bugs. In JabRef we detected two bugs only with the configuration EDG-2, and one bug with the configuration EFG-2 and EDG-2. In Rachota we detected one bug only with the configuration EDG-2. In the following we sketch one bug detected in JabRef using our approach. For a detailed exposition we have set up a website containing supporting material of this paper; see Section VIII. In JabRef the following test sequence 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`; (3) switch back to the dialog and click `OK`. The error occurs, because the newly opened dialog starts *modeless* which allows the user to close the database, although the dialog still allows the user to modify the database. Note that JabRef does not show an error message. Instead, the stack trace of the `ArrayOutOfBoundsException` is printed on the standard output (`stdout`). We reported all detected bugs to the corresponding developers. Furthermore, all bugs have been fixed in the following version of the applications.



(a) EFG (b) EDG

Figure 16. Snippets of the EFG and EDG of JabRef. The edges marked in red in the EFG and in the EDG are used to generate the test sequence that detects the bug.

Figure 16 shows a snippet of the EFG and the EDG of JabRef that corresponds to the detected bug. In the test sequence generation for event `Close database` our approach detects the write/read dependency to event `OK`. This dependency consists of a field for JabRef's metadata, which is written in `Close database` and read in `OK`. Thus, the event dependency sequence ⟨`Close database`, `OK`⟩ is generated. This event dependency sequence is transformed into a test sequence, since there exists no corresponding path in the EFG. The shortest path from an initial event to `Close database`, and from `Close database` to `OK` is computed and inserted to the sequence. The resulting test sequence is: ⟨`Manage content selectors`, `Close database`, `Manage content selectors`, `OK`⟩. The EFG-2 approach does not detect this bug.

V. Threats to Validity

The first threat to internal validity is the step GUI Ripper. Since this step represents a dynamic approach (i.e., the application is executed in order to extract events), it cannot be guaranteed to find all widgets of the application. For instance, the application itself might be hostile or even faulty, e.g., if the GUI opens a new window in the background, the GUI ripper is not able to find it, and thus, it cannot be considered during EFG construction. Furthermore, the fact if a widget is enabled or disabled during ripping may strongly depend 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. Hence, the EFG obtained from the GUI ripper represents an approximation of the application's event flow. It cannot be guaranteed that a path in the constructed EFG is executable on the GUI; the events in the EFG are yet pairwise executable.

The second threat to internal validity is the replication of the experiments. All applications store user settings to the hard disk, e.g., enabled and disabled toolbars, and recently opened files. 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. Otherwise test cases may mistakenly fail, e.g., a widget is not found due to an existing user setting.

The third threat to internal validity is that the applications are strongly connected to the date and time in the moment of their execution. When replaying the test cases, some of them may fail, because widgets are not recognized anymore (while replaying a *calendar control* shows a different date as the calendar control was ripped). In order to decrease the threats to internal validity we ran the experiments three times.

One threat to external validity is the portability of our approach. We evaluated four Java applications which incorporate the Swing toolkit[6] for building the user interface. Alternative toolkits, e.g., the Standard Widget Toolkit (SWT)[7], follow different paradigms of building a user interface, i.e., using native widgets written in C, instead of pure Java widgets. Thus, the construction of the black-box model and the implementation of the static analysis must be adapted to the corresponding environment. Under the assumption that this adaption is possible, our approach can be extended to those platforms.

VI. Discussion

In this section we discuss the steps *Event Handler Extraction*, *Static Analysis*, and *Test Sequence Generation* of our approach.

A. Event Handler Extraction

The step Event Handler Extraction represents a dynamic approach (i.e., it executes a GUI application in order to extract event handlers). In principle, it would be possible to extract these event handlers in a static approach, e.g., by analyzing the source code. However, since Java code is written in so many ways, a static analysis technique must be tailored to comprehend how a GUI is built. For example, event handlers might also be registered using callbacks, virtual function calls, or even external resource files.

B. Static Analysis

When constructing the EDG, the step Static Analysis does not consider potential aliasing of fields or potentially infeasible control-flow. Furthermore, Java distinguishes between instance fields and class fields, which are treated the same way in our bytecode analysis. That is, both class fields and instance fields are mapped to their corresponding class. Moreover, instance fields are not mapped to their objects. The static analysis does not distinguish between calls of instance methods and class methods and thus, is not reliable regarding polymorphism. Hence, the resulting EDG is an approximation of the dependencies between fields. However, we are interested in prioritizing events, so a lightweight static analysis is sufficient while leaving room for further in-depth analyses.

[6]http://docs.oracle.com/javase/tutorial/uiswing/index.html

[7]http://www.eclipse.org/swt/

C. Test Sequence Generation

The step Test Sequence Generation transforms each event dependency sequence into a test sequence by finding the shortest paths in the EFG as explained in Section III. Thus, for a set of test sequences, the same prefix and suffix path in the EFG could be used several times. However, this does not present a drawback for two reasons: (1) If the prefix or the suffix path consists of a *relevant* event sequence, then this sequence will be separately considered as an event dependency sequence (and later on as a test sequence). (2) If the prefix or the suffix path consists of an *irrelevant* event sequence, then this sequence simply serves for the purposes of making an event dependency sequence executable. However, a possible extension to the step Test Sequence Generation is to consider *unused* pre- and suffix paths.

VII. Related Work

In a broad sense, our approach is related to the generation of sequences of method calls. The work that comes closest to our work, described in [23], uses a call sequence and a static analysis in order to generate relevant sequences of method calls for Java objects. Our work on event sequence generation differs in two main aspects. First, we consider dependencies of method calls across unit boundaries (for the goal of system testing, as opposed to unit testing as in [23]); for example, we analyze the dependent libraries of a GUI application. Second, we accommodate the requirement of executability (and not just the requirement of relevance, as in [23]), namely by incorporating the use of a black-box model.

In [21], the GUI run-time feedback is obtained from the execution of a "seed test suite". The feedback is used to iteratively generate new improved test cases. However, a test suite with events of reasonable length has to be generated and executed before building new improved event sequences based on a pure black-box model. Our approach does not need a "training set" since it immediately selects relevant event sequences using a static analysis once a black-box model is provided. AutoBlackTest [11] executes an AUT and uses reinforcement learning to obtain relevant event sequences. EXSYST [8] observes which events correspond to a certain behavior in the source code. In contrast with the three previously cited approaches, our approach executes the AUT only for constructing the black-box model of the GUI, and *not* for generating event sequences.

VIII. Conclusion and Future Work

In this paper we have proposed a new approach to select relevant event sequences among the event sequences generated by a black-box model. We express the relevance of an event sequence by a precisely defined dependency between a fixed number of events in the event sequence.

We have implemented the approach. Our experiments have the following findings.

- Our approach scales to realistic GUI applications.
- Our approach discards an interesting number of irrelevant event sequences.
- Our approach generates test sequences that would be generated in the pure black-box approach only with very high values for the parameter (and thus with very high cost).
- Our approach achieves the same coverage as the pure black-box approach, even in the cases when the number of test cases is smaller.
- Our approach detects previously undetected bugs.

An interesting line of research for future work is to explore whether a dynamic approach such as [1] or [6], which may help with the generation of 'relevant' input data for test cases, can be integrated with our approach to select relevant test sequences.

Instead of generating and replaying all test sequences at once, we plan to integrate an iterative approach such as [8] or [11]. The benefit is that one can additionally bound the cost of the testing process by defining a specific timeout.

We plan to incorporate an approach such as [22] to generate event sequences for multi-threaded GUI applications. For example, in e-Mail applications a user interaction may initiate several threads (e.g., fetching e-mails from different accounts) that could lead to *invalid thread access* errors.

We have set up a website containing supporting material for this paper:

http://www.informatik.uni-freiburg.de/~arlt/issre2012

ACKNOWLEDGMENTS

This work is partially supported by the research projects EVGUI, ARV, and SAFEHR (funded by the Macau Science and Technology Development Fund and the Chinese NSFC No. 61103013), and the US National Science Foundation under grant CNS-1205501 and CNS-0855055. Cristiano Bertolini was partially supported by CAPES-PNPD/2011.

REFERENCES

[1] S. Arlt, P. Borromeo, M. Schäf, and A. Podelski. Parameterized GUI Tests. In *ICTSS*, 2012.

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

[3] F. Belli, M. Linschulte, C. J. Budnik, and H. A. Stieber. Fault Detection Likelihood of Test Sequence Length. In *ICST*, pages 402–411, 2010.

[4] C. Bertolini, A. Mota, and E. Aranha. Calibrating Probabilistic GUI Testing Models Based on Experiments and Survival Analysis. In *ISSRE*, pages 319–328, 2010.

[5] C. Bertolini, A. Mota, E. Aranha, and C. Ferraz. GUI Testing Techniques Evaluation by Designed Experiments. In *ICST*, pages 235–244, 2010.

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

[7] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[8] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *ISSTA*, pages 67–77, 2012.

[9] S. Huang, M. B. Cohen, and A. M. Memon. Repairing GUI Test Suites Using a Genetic Algorithm. In *ICST*, pages 245–254, 2010.

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

[11] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In *ICST*, pages 81–90, 2012.

[12] A. M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.*, 17(3):137–157, 2007.

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

[14] A. M. Memon, I. Banerjee, and A. Nagarajan. What Test Oracle Should I Use for Effective GUI Testing? In *ASE*, pages 164–173, 2003.

[15] A. C. R. Paiva, J. C. P. Faria, and P. M. C. Mendes. Reverse Engineered Formal Models for GUI Testing. In *FMICS*, pages 218–233, 2007.

[16] J. C. Silva, C. E. Silva, R. D. Gonçalo, J. Saraiva, and J. C. Campos. The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In *EICS*, pages 181–186, 2010.

[17] T. Tuglular, C. A. Muftuoglu, F. Belli, and M. Linschulte. Event-Based Input Validation Using Design-by-Contract Patterns. In *ISSRE*, pages 195–204, 2009.

[18] L. J. White and H. Almezen. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *ISSRE*, pages 110–123, 2000.

[19] L. J. White, H. Almezen, and N. Alzeidi. User-Based Testing of GUI Sequences and Their Interactions. In *ISSRE*, pages 54–65, 2001.

[20] X. Yuan, M. B. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated gui testing. In *ASE*, pages 405–408, 2007.

[21] X. Yuan and A. M. Memon. Using GUI Run-Time State as Feedback to Generate Test Cases. In *ICSE*, pages 396–405, 2007.

[22] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded GUI applications. In *ISSTA*, pages 243–253, 2012.

[23] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, 2011.