Parameterized GUI Tests

Stephan Arlt[1], Pedro Borromeo[1], Martin Schäf[2], and Andreas Podelski[1]

[1] Albert-Ludwigs-Universität Freiburg
[2] United Nations University Macau

Abstract. GUI testing is a form of system testing where test cases are based on user interactions. A user interaction may be encoded by a sequence of events (e.g., mouse clicks) together with input data (e.g., string values for text boxes). For selecting event sequences, one can use the black-box approach based on Event Flow Graphs. For selecting input data, one can use the white-box approach based on parameterized unit tests and symbolic execution. The contribution of this paper is an approach to make the principle of parameterized unit testing available to black-box GUI testing. The approach is based on the new notion of *parameterized GUI tests*. We have implemented the approach in a new tool. In order to evaluate whether parameterized GUI tests have the potential to achieve high code coverage, we apply the tool to four open source GUI applications. The results are encouraging.

1 Introduction

GUI testing is a form of system testing where test cases are based on user interactions. A user interaction may be encoded by a sequence of events (e.g., mouse clicks) together with input data (e.g., string values for text boxes). For selecting event sequences, one can use a black-box approach based, e.g., on EFGs (Event Flow Graphs, [9]). For selecting input data, one can use a white-box approach based, e.g., on parameterized unit tests [14] and dynamic symbolic execution [3].

Motivated by the established success of the black-box approach to GUI testing [2, 9, 16], we ask the question whether the black-box approach can be integrated with techniques from the white-box approach so that the resulting approach provides both, the selection of event sequences and the selection of input data.

Given the established success of parameterized unit testing [3, 4, 6, 14, 15], and given the apparent analogy between event handlers called in a GUI test and methods called in a parameterized unit test, it seems natural to ask whether we can obtain the desired integration by replacing method calls with event handler calls. At first sight, this approach is not possible: the assignment of the input data (e.g., the string value filled in by the user in a text box) cannot be found in any event handler called in a GUI test (the assignment is done, letter by letter, in the *message loop* of the GUI toolkit). There are other, more technical obstacles (event handlers call native code of the GUI toolkit, event handlers hold a private

access modifier which makes them unavailable for symbolic execution, etc.). I.e., the naive approach does not work.

The contribution of the work presented in this paper is an approach to make the principle of parameterized unit testing available to black-box GUI testing. The approach is embodied in a new tool, called *Gazoo*. Gazoo selects event sequences from the EFG of a GUI application and generates a set of *parameterized GUI tests*. Then, Gazoo applies Pex [3] in order to instantiate the parameterized GUI tests. Finally, Gazoo replays instantiated GUI tests on the GUI application.

In the terminology of the black-box/white-box dichotomy, Gazoo starts with a black-box approach (using the EFG in order to select executable test sequences), then moves on to a white-box approach (in order to generate parameterized GUI tests and instantiate them using Pex), and finally goes back to the black-box approach (using a replayer in order to execute the (instantiated) GUI tests on the GUI application). To establish the appropriate interface between the black-box approach and the white-box approach, we need to overcome a number of technical hurdles. In particular, we build an instrumented version of the GUI application in order to extract *sequential programs* as used in the parameterized unit test. We replace GUI widgets by *symbolic widgets* and inject *symbolic events* into the sequential programs in order to obtain what we call a *parameterized GUI test*. We evaluate Gazoo on four open source GUI applications. The experimental results indicate that parameterized GUI tests have the potential to achieve high code coverage.

2 Example

We illustrate how our approach tests GUI applications using an over-simplified example application given in Figure 1. The example application provides the functionality of an address book. The main window consists of two buttons that can add or remove a contact. When clicking the *add* button, a dialog window appears which provides two text boxes, for the first name and for the last name, and two buttons to store (OK) or discard (Cancel) the contact. In the following, we use the term *application* to refer to a *GUI application*.

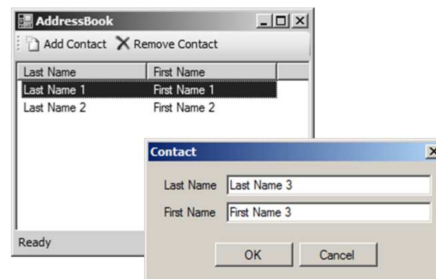

Fig. 1. Screen shot of the example application. The AddressBook application consists of two windows, a main window and a dialog. Clicking on `Add Contact` opens a new dialog and disables the events of the main window. Clicking on `OK` or `Cancel` closes the dialog and re-enables the events of the main window.

2.1 Selecting Test Sequences

When testing applications through its GUI there exist different possibilities in which order to interact with widgets. For example, one can first click the *remove* button and then the *add* button. However, the reverse order does not work as clicking the *add* button opens the dialog window, so *remove* cannot be clicked until the dialog window is closed. Thus, not all sequences of events are executable on the application. In order to avoid those non-executable event sequences, our approach incorporates a black-box model of the GUI, the Event Flow Graph (EFG) [9] depicted in Figure 2.

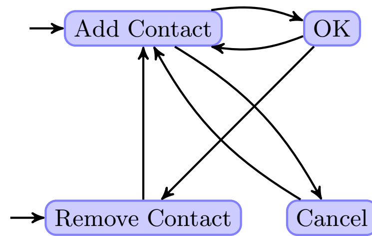

Fig. 2. Event Flow Graph of the example application. The events `Add Contact` and `Remove Contact` represent initial events that can be executed immediately after the application is launched. In contrast, the events `OK` and `Cancel` can be executed not until `Add Contact` is triggered.

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 of the GUI. Each event in $I \subseteq E$ is an initial event which can be executed immediately after the application is launched. An edge $(e, e') \in \delta$ between two events $e, e' \in E$ states that the event e' can be executed after the event e. If there is no edge between events e, e' then event e' cannot be executed after event e.

Using the EFG one can generate a set of *event flow sequences* of the application. An event flow sequence is a walk of a specific length in the EFG. Throughout this paper we generate event flow sequences of length 2, that is, neighbors of events. Event flow sequences do not necessarily start in an initial event, and thus, are not executable on the application. We expand event flow sequences to *test sequences* by inserting the shortest path from the first event of the event flow sequence to an initial event of the EFG. A test sequence $s = e_0, \ldots, e_n$ is a sequence of events, such that $e_0 \in I$ and $(e_i, e_{i+1}) \in \delta$ for all $0 \leq i < n$. Hence, a test sequence starts with an initial event and is executable on the application. Figure 3 shows all resulting test sequences of the example application obtained by event flow sequences of length 2 from the EFG. For the example application, our approach generates 6 test sequences in total.

The benefit of the EFG is the possibility to generate test sequences which are executable on the application. However, test sequences do not account for input data to widgets. When executing a test sequence on a GUI, recent efforts [9, 17, 18] insert random input data to widgets. We believe that the choice of input data is both vital to the coverage that can be achieved, and to the total number of executed tests. For example, choosing random values can result in low coverage.

Fig. 3. Test Sequences of the example application. The dark-colored events represent the events of the event flow sequence of length 2. The light-colored events represent intermediate events that make the event flow sequence executable on the GUI of the application.

$t_1 = \langle$ `Add Contact`, `OK` $\rangle$
$t_2 = \langle$ `Add Contact`, `Cancel` $\rangle$
$t_3 = \langle$ `Remove Contact`, `Add Contact` $\rangle$
$t_4 = \langle$ `Add Contact`, `OK`, `Add Contact` $\rangle$
$t_5 = \langle$ `Add Contact`, `OK`, `Remove Contact` $\rangle$
$t_6 = \langle$ `Add Contact`, `Cancel`, `Add Contact` $\rangle$

Furthermore, multiple random values can result in a prohibitive large number of test cases (e.g., one randomly chosen value is integrated in one test case).

2.2 Generating Parameterized GUI Tests

To enable the generation of input data for test sequences, we introduce *Parameterized GUI Tests* which are test sequences parameterized by possible input data. In the following we first outline how parameterized GUI tests are generated. Then we describe how input data to parameterized GUI tests is generated in our approach.

```
1  class AddressBookWindow {
2      private ListView contacts;
3
4      // handler for event "Add Contact"
5      private void OnAddContact() {
6          ContactDialog dialog = new ContactDialog();
7          dialog.ShowDialog(this);
8      }
9
10     class ContactDialog {
11         private TextBox lastName;
12
13         // handler for event "OK"
14         private void OnOK() {
15             if ( 0 == lastName.Text.Length ) {
16                 return;
17             }
18             if ( lastName.Text.Length > 255 ) {
19                 throw new Exception("Text is too long.");
20             }
21             contacts.AddItem(lastName.Text);
22         }
23     }
24 }
```

Fig. 4. Excerpt of the source of the example application. The source code consists of two classes (`AddressBookWindow` and `ContactDialog`) and three event handlers (`OnAddContact`, `OnRemoveContact`, and `OnOK`). The event handler `OnOK` evaluates the text of the text box `lastName`. A new contact is only added, if the last name is not empty and contains less than 256 characters.

Figure 4 shows an excerpt of the underlying source code of the example application. The method `OnAddContact` represents the event handler which is

$p_1 = \langle$Add Contact, lastName, OK$\rangle$
$p_2 = \langle$Add Contact, Cancel$\rangle$
$p_3 = \langle$Remove Contact, Add Contact$\rangle$
$p_4 = \langle$Add Contact, lastName, OK, Add Contact$\rangle$
$p_5 = \langle$Add Contact, lastName, OK, Remove Contact$\rangle$
$p_6 = \langle$Add Contact, Cancel, Add Contact$\rangle$

Fig. 5. Parameterized GUI tests of the example application. In the PGT p_1, p_4, p_5, the event `OnOK` is prefixed with the parameter `lastName`. The PGTs s_2, s_3, s_6 do not need parameters.

$t_{1a} = \langle$AddContact, *empty string*, OK$\rangle$
$t_{1b} = \langle$AddContact, *''a string with more than 255 characters''*, OK$\rangle$
$t_{1c} = \langle$AddContact, *''Last Name 3''*, OK$\rangle$

Fig. 6. Instantiated GUI tests of the parameterized GUI test p_1. The GUI test t_{1a} contains as input data an empty string; t_{1b} contains a string with a length greater than 255; and t_{1c} contains a string with a length lesser than 255.

executed once the corresponding button in the main window is clicked. The method `OnOK` is executed if the OK button in the dialog window is clicked. The event handler `OnOK` adds an element to the list of `contacts` (line 21), if the text of the text box `lastName` is not empty and contains less than 256 characters. If the text is empty, the event handler returns without adding a contact (line 16). If the text is too long, it returns with an exception (line 19).

Our approach detects that event handler `OnOK` evaluates the text of the text box `lastName` in the conditions (line 15 and line 18). That is, the event handler `OnOK` might need input data. We transform the test sequences from Figure 3 into parameterized GUI tests depicted in Figure 5. In particular, we prefix the event `OnOK` with the parameter `lastName`. The parameter `lastName` can adopt different values which can lead to different execution paths in the event handler `OnOK`. In our example application, only one text box is evaluated. If further text boxes are evaluated, we add parameters to the test sequence for each of them. Our approach does not only consider input data to text boxes, as described in Section 3.

Our approach instantiates each parameterized GUI test by an automatic computation of suitable input data. In this paper we incorporate Pex [3]. In general, input data can also be provided by alternative tools [15]. Pex uses dynamic symbolic execution to identify sets of input values that execute all control-flow paths of the program in the parameterized GUI test. E.g., for the parameterized GUI test $p_1 = \langle$Add Contact, lastName, OK$\rangle$, Pex identifies three distinct values of `lastName` that have to be tested as shown in Figure 6. With the valuations for `lastName` and the parameterized GUI test, we have all ingredients for a GUI test which can be executed using our replayer. The replayer accepts

a set of GUI tests and mimics the events (user interactions) on the application. If a GUI test contains input data, this data is transferred to the corresponding widget. Furthermore, the replayer integrates an oracle that determines whether a GUI test passed or failed.

3 Approach and Implementation

In this section we present details of our approach and its implementation. As outlined in Section 1, there exist a bunch of issues in order to make the approach of parameterized GUI tests applicable to real world applications. Our approach depicted in Figure 7 consists of the following consecutive steps: (1) Event Flow Construction; (2) Symbolic Widget Injection; (3) Symbolic Event Injection; (4) Event Handler Elevation; (5) Generation of Parameterized GUI Tests, (6) Symbolic Execution, and (7) Replayer.

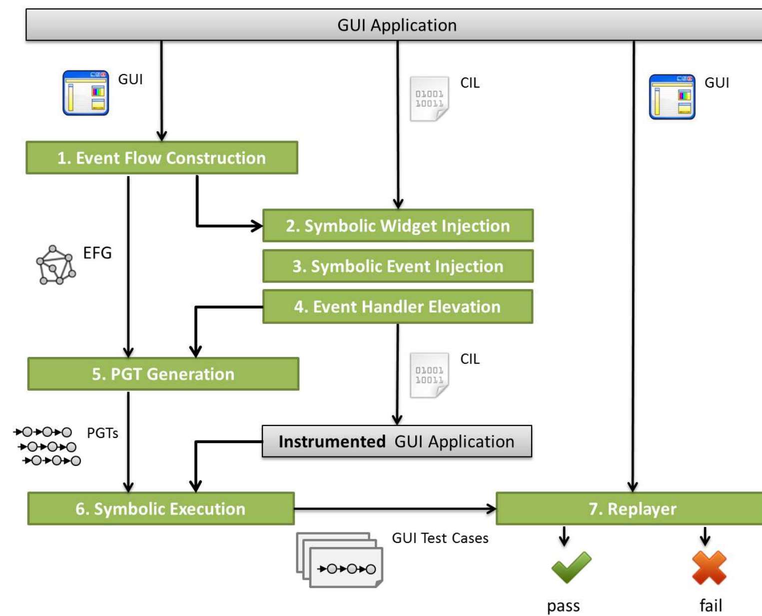

Fig. 7. Our approach, consisting of seven consecutive main steps. The input to our approach is a *GUI Application*. Input data is generated on the *Instrumented GUI Application*. GUI tests are replayed on the original *GUI Application*.

3.1 Event Flow Construction

The starting step of our approach is the *Event Flow Construction*. It takes the GUI of an application as input and outputs an Event Flow Graph. First, we exe-

cute the application and record its *GUI structure*. Second, we construct the EFG from the GUI structure. A GUI structure consists of widgets (e.g., windows, buttons, text boxes) and their corresponding properties (e.g., enabled or disabled). While executing the application, we enumerate all widgets of the GUI. This is done by calling specific functions provided by the GUI toolkit. For each found widget (e.g., a button) we trigger the assigned event (i.e., a click). If the click on the button opens a new window, we continue to record the GUI structure of the recently opened window and so on. The process stops if all found windows have been explored. Since a GUI represents a hierarchical structure, a depth-first search is performed. The obtained GUI structure is transformed into an Event Flow Graph. 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 details of the EFG construction can be found in [10].

In our approach we enhance the EFG construction, such that, for each widget the event handler assigned to this widget is additionally stored in the GUI structure. This information is later needed during the generation of parameterized GUI tests and during the replaying of GUI tests.

3.2 Symbolic Widget Injection

In our approach we want to generate suitable input data, e.g., we want to reason about string values of text boxes. However, in order to perform a symbolic execution, we have to replace regular widgets by symbolic widgets. There are two main reasons: First, a change to a regular widget's property leads to a native call to the GUI toolkit. Including code of the GUI toolkit in the analysis is usually not feasible, as it would significantly increase the size of the code that has to be analyzed. Furthermore, in many cases, the code is in native format and thus not accessible by the analysis. Second, our approach focuses on validating the behavior of an application. In particular, we are not interested in validating the behavior of the GUI toolkit, i.e., validating whether a redraw of a widget was successful.

The step *Symbolic Widget Injection* takes the CIL[3] code of an application as input and replaces widgets by symbolic widgets. Figure 8 shows an excerpt of the symbolic representation of a text box. Gazoo uses Microsoft CCI[4] to modify the CIL code. By default, the main widgets included in the Windows Forms framework are considered, e.g., text boxes, check boxes, radio buttons etc. *Gazoo* is highly configurable: One can define further symbolic widgets for alternative GUI toolkits, such as Silverlight.

3.3 Symbolic Event Injection

In GUI applications, specific events do not have their own event handlers. For example, it is not likely to have an event handler which assigns a string value to a

[3] http://msdn.microsoft.com/en-us/netframework/aa569283.aspx

[4] http://ccimetadata.codeplex.com/

```
1  class TextBox {
2
3    public string Text {
4      get {
5        // native call
6        return GetWindowText();
7      }
8      set {
9        // native call
10       SetWindowText(value);
11     }
12   }
13 }
```

```
1  class SymbolicTextBox {
2
3    string text;
4
5    public string Text {
6      get {
7        // a "getter"
8        return this.text;
9      }
10     set {
11       // a "setter"
12       this.text = value;
13     }
14   }
15 }
```

Fig. 8. Comparison of regular widgets (left) and symbolic widgets (right). In symbolic widgets, native calls to the GUI toolkit are pruned (line 8 and line 12). A symbolic representative of the widget property (line 3), i.e., `text`, is injected. This property can be read and written by the `get` and `set` operations.

text box, once a user presses a key on the keyboard. This behavior is implemented in the GUI toolkit and does not exist in the application itself. In order to assign a string value, generated by the symbolic execution, to a text box, our approach injects symbolic events to the application. That is, we partially re-implement the event handlers of the GUI toolkit.

The step *Symbolic Event Injection* takes CIL code with symbolic widgets as input and returns a modified version of the CIL code, including symbolic widgets and symbolic events (see Figure 9). *Gazoo* visits the instructions of the CIL code. If it encounters an evaluation of a widget property, e.g., the `Text` property of a text box is evaluated, a symbolic event is added to the CIL code. A symbolic event is a *setter* method that takes one parameter representing the value to be assigned to the corresponding widget property. In our approach we separate the concerns of having both symbolic widgets and symbolic events: Symbolic widgets address the issue that properties of GUI widgets imply native calls. In contrast, symbolic events provide an interface that allows to assign a value to a widget property. Furthermore, in our setting we can assign values to widget properties. However, there exist widgets that prohibit the assignment of arbitrary property values. For example, the property `Count` which indicates the number of items in a *list widget*. In order to change the property `Count`, one has first to add an item to the list widget which increments the property `Count`. Those complex symbolic events are out the scope of this work and will be addressed in a future work.

3.4 Event Handler Elevation

Usually, in programming languages like C#, event handlers are implemented as *private* methods within classes. They are only visible to their surrounding class, and thus, cannot be called directly. In order to allow an exploration of the event handlers by the symbolic execution, we *elevate* event handlers. That is, for

```
1  // regular text box
2  private TextBox lastName;
3
4  // regular event handler
5  private void OnOK() {
6    if (0 == lastName.Text.Length)
7    {
8      // ...
9    }
10 }
```

```
1  // symbolic text box
2  public SymbolicTextBox lastName;
3
4  // elevated event handler
5  public void OnOK() {
6    if (0 == lastName.Text.Length)
7    {
8      // ...
9    }
10 }
11
12 // symbolic event
13 public void SetLastNameText(string
       text) {
14   lastName.Text = text;
15 }
```

Fig. 9. Comparison of code from the original application (left), and the instrumented application (right). The instrumented application contains symbolic widgets (line 2), symbolic events (line 13), and elevated event handlers (line 5).

each method of the application we change their access modifiers from *private* to *public*; see Figure 9.

Gazoo visits the classes and methods of the executable. If it encounters a private class or a private method, it changes the access modifier and serializes all changes to the CIL code. Note that *Gazoo* also visits and modifies classes, in case they are not visible. The output of these steps is a *valid* executable. In particular, elevating classes and methods do not raise conflicts. For example, the access modifier does not influence the unique signature of a method.

3.5 Parameterized GUI Test Generation

Having obtained the EFG of a GUI (step 1) and built an instrumented version of the application (steps 2, 3, and 4), Gazoo generates a set of parameterized GUI tests. This step consists of two sub-steps:

First, Gazoo generates test sequences of a specific length from the EFG. Each test sequence represents a program that sequentially calls the event handlers of the events in the sequence. Second, for each event in the test sequence, Gazoo analyzes whether the event handlers rely on input data. For example, an event handler evaluates the property of a widget. If so, Gazoo transforms the test sequence into a parameterized GUI test. For each evaluated widget property, we add a new parameter to the parameterized GUI test. Furthermore, we prefix the event handler (that relies on input data) with a call to the symbolic event that assigns the input data. The idea is that the symbolic event writes the input data, while the selected event handler evaluates the input data. Figure 10 shows the difference between a test sequence and a parameterized GUI test.

3.6 Symbolic Execution

Having generated a set of parameterized GUI tests, our approach instantiates each parameterized GUI test by applying Pex. Pex takes as input a parame-

```
1 // a test sequence
2 void TestSequence()
3 {
4   OnAddContact();
5   OnOK();
6 }
```

```
1 // a parameterized GUI test
2 void PGT(string lastname)
3 {
4   OnAddContact();
5   SetLastNameText(lastName);
6   OnOK();
7 }
```

Fig. 10. Comparison of a test sequence (left) and a parameterized GUI test (right). In the parameterized GUI test, the call of event handler `OnOK` is prefixed with the symbolic event `SetLastNameText`. This symbolic event sets the parameter value `lastname` of the PGT to a text box.

terized test and performs a dynamic symbolic execution on the instrumented application. The output of Pex is a set of concrete values of the parameters in the parameterized test. For each element in this set, we create an instantiated GUI test. An instantiated GUI test consists of the sequence of events from the parameterized GUI test, and the concrete parameter values for widget properties.

3.7 Replayer

The last step of our approach is the *Replayer*. The replayer takes as input a set of instantiated GUI tests and replays them on the original application. First, the replayer launches the application. Then, it executes the instantiated GUI test, consisting of an event sequence and its concrete parameter values for widget properties. After replaying a GUI test, the replayer closes the application. In our setting, the replayer uses a crash monitor as the oracle for each instantiated GUI test. However, the replayer is able to adopt further test oracles [11].

For each event handler in the GUI test, the replayer looks up the corresponding event in the EFG. Moreover, the replayer looks up the associated widget of the event in the GUI structure. Using this information, the replayer can find the widget on the GUI and can trigger its corresponding event. Gazoo incorporates *Ranorex*[5] to mimic user interactions, encoded as events, on the application.

For each parameter value in the GUI test, the replayer looks up the intended widget property. As described above, each parameter in the parameterized GUI tests is associated to one symbolic event. Moreover, each symbolic event writes a specific property of a widget. Like for the events in the GUI test, the replayer finds the widget on the GUI using the EFG and its GUI structure. Then, the replayer assigns the value of a parameter to the corresponding widget. This is done via *Reflection* and *Memory-mapped files*[6] in order to send data across processes (i.e., the replayer and the application under test). In Section 5 we discuss the implication of using reflection and memory-mapped files in GUI testing.

[5] http://www.ranorex.com/
[6] http://msdn.microsoft.com/en-us/magazine/cc163617.aspx

	AddressBook	OpenImage	Handbrake	FareCalculator
LOC	2778	2347	520	298
Classes	98	87	30	22
Methods	163	109	19	14
Events	45	13	7	3

Fig. 11. Statistical data of the AUTs used in our experiments.

4 Experiments

In this section we evaluate our approach. We compare how our approach performs, (a) when the computation of input data is replaced by the use of random values, and (b) when the Event Flow Graph is not considered for event sequence generation. We first present the setup of the experiments. Then we discuss the results of the experiments. We define the following two research questions:

- **Q1**: Is it reasonable to use Pex-generated values instead of random values for widgets? A priori, this is not clear, for two reasons: (1) In GUI applications, events that evaluate input data might be simple, that is, they only check whether an input is entered or not. Then, one can achieve a reasonable coverage by providing arbitrary input (or no input). (2) Events might evaluate input data in complex ways, that is, checking whether a specific string is entered or not. Then, one cannot achieve a reasonable coverage due to limitations of the symbolic execution (wrt. to the underlying constraint solver).
- **Q2**: Is it reasonable to incorporate the Event Flow Graph in order to generate parameterized GUI Tests? In principle, the idea of selecting event sequences of an application is related to the generation of method calls of a library. In libraries, one can call each method at any time. Hence, there exist no order, in which library methods are allowed to call. In GUI applications one can call an event handler at any time as well. However, a call of an event handler may not be allowed, e.g., when the window of an event handler is not yet displayed. This leads to GUI tests that are not executable on the GUI.

4.1 Setup of the Experiments

We evaluate our approach on four C# open source applications: *AddressBook* manages contacts; *OpenImage* downloads images from websites; *HandBrake Encoder* converts video files; *FareCalculator* calculates ticket prices for trains. Except for FareCalculator [5], all other applications are fetched from CodePlex[7]. It is important to observe that we use stable versions where bugs are rarely found. We choose various applications to cover different code styles. Figure 11 shows some statistics of each AUT (Application Under Test).

Our experiments consists of the three configurations **A**, **B**, **C**. The configuration **A** generates event sequences of length 2 from the EFG, and uses Pex

[7] http://www.codeplex.com/

to generate input values for the event sequences. The choice of the parameter 2 is motivated by previous empirical studies on bugs in GUI applications [17]. The configuration **B** generates event sequences of length 2 from the EFG, but uses random values as input data. In order to have statistical confidence, we choose random values using 10 different seeds. Thus, each parameterized GUI tests is instantiated 10 times containing different input data. The configuration **C** generates all sequences of events of length 2. That is, it does not use the EFG, and thus, might select non-executable event sequences. By comparing configuration A and B, we investigate the coverage that our approach can achieve. By comparing configuration A and C, we investigate the number of non-executable GUI tests that our approach discards.

As a precondition of all GUI tests we define that all user settings of an AUT have to be deleted before executing the GUI test. As a postcondition of all GUI tests we use a crash monitor. In particular, we record any exception occurred during test case execution, and we automatically observe if a test case is executable on the GUI. For a discussion of alternative oracles we refer to [4, 11].

The GUI tests are executed on 10 virtual Windows machines with 2.0 GHz CPU, 2 GB RAM, 500 GB HDD. In order to mitigate the effect of randomness, the configurations A, B and C are executed three times. The total number of executed test cases amounts to 24,063.

4.2 Results of the Experiments

Figure 12 shows the results of the experiments. We answer **Q1** with **Yes**: We find that it is reasonable to use Pex-generated input values instead of random input values. In all AUTs, the configuration A achieves a higher line and a higher branch coverage than the configuration B. For OpenImage, the improvement of the line coverage amounts to 19%, for AddressBook 41%, and for HandBrake 45%. FareCalculator is an outlier; the line coverage improvement is 76%. The reason is that FareCalculator consists of event handlers that need specific input data. Pex is able to generate this input data, while random values do not suffice. It is unlikely to achieve 100% line and branch coverage in an application, as the applications may also need input data that cannot be generated automatically. For example, if an application requires a valid URL to download an image from the web, Pex cannot generate such a valid URL. In this case, the application depends on external test data that must be specified by a test engineer.

We answer **Q2** with **Yes**: We find that it is reasonable to incorporate the Event Flow Graph in order to generate parameterized GUI Tests. For AddressBook, the configuration A generates 319 PGTs which leads to 349 instantiated GUI tests. In comparison, the configuration C generates 2025 PGTs which leads to 2352 instantiated GUI tests. Thus, 2003 out of 2352 GUI tests, that is 85%, are not executable on the application. For OpenImage, 17% of the GUI tests are not executable on its GUI. The reason is that in AddressBook and OpenImage it is not allowed to execute an arbitrary event at any time. For the AUTs HandBrake and FareCalculator, the configuration C generates the identical set

AUT / Configuration	A	B	C
AddressBook			
Line Coverage (%)	**74**	**43**	74
Branch Coverage (%)	**65**	**38**	65
# PGTs	319	319	2025
# GUI Tests	349	3190	2352
Generation Time (s)	407	255	2739
Execution Time (m)	**93**	850	**730**
# Non-executable GUI Tests	**-**	**-**	**2003**
OpenImage			
Line Coverage (%)	**63**	**51**	63
Branch Coverage (%)	**59**	**29**	59
# PGTs	139	139	169
# GUI Tests	148	1390	179
Generation Time (s)	278	222	336
Execution Time (m)	**38**	359	**46**
# Non-executable GUI Tests	**-**	**-**	**31**
HandBrake			
Line Coverage (%)	**88**	**48**	88
Branch Coverage (%)	**84**	**44**	84
# PGTs	49	49	49
# GUI Tests	73	490	73
Generation Time (s)	71	42	71
Execution Time (m)	**17**	116	**17**
# Non-executable GUI Tests	**-**	**-**	**-**
FareCalculator			
Line Coverage (%)	**93**	**22**	93
Branch Coverage (%)	**91**	**19**	91
# PGTs	9	9	9
# GUI Tests	39	90	39
Generation Time (s)	49	34	49
Execution Time (m)	**8**	20	**8**
# Non-executable GUI Tests	**-**	**-**	**-**

Fig. 12. Results of the experiments.

of PGTs as configuration A. In these applications, the EFG is fully-connected, and each event is also an initial event. We believe that is reasonable to incorporate the EFG by default: For large applications, our approach generates a subset of event sequences of the GUI. The event sequences in this subset are actually executable on the GUI. For small applications, our approach generates the same set of event sequences which would be generated without considering the EFG.

4.3 Threats to Validity

Beyond the selection bias due to the limited availability of open source C# applications, we report one threat to external validity: We evaluated four C# open source applications which incorporate the Windows Forms toolkit for building the GUI. Alternative programming languages and GUI toolkits, e.g., Java Swing, follow different paradigms of building graphical user interfaces. For example, it might be not possible to obtain event handlers during the construction of the

EFG. Thus, the construction of the EFG, the generation of parameterized GUI tests, and the symbolic execution must be adapted to the corresponding environment. In principle, there is no reason to believe that our approach is not applicable to other environments.

5 Discussion

Why a black-box model? In this paper we use a black-box model to represent events and their corresponding event flow. An EFG is constructed by executing the application and observing the behavior of its GUI. In principle it is also possible to use a white-box model of the application. For example, this white-box model might be constructed by techniques from static analysis. Since GUI code is written in many ways, a static analysis technique must be tailored to comprehend how a GUI is built. The use of a black-box model is justified by the reasonable trade-off between applicability and precision. The constructed EFG in our approach represents an approximation of the actual event flow of the application. Thus, our approach cannot guarantee to find all events of the application. For example, the application itself might be hostile or even faulty.

Why a replayer? One can argue it is not necessary to replay instantiated GUI tests on the original application. For example, one can execute GUI tests in a fashion of unit testing by simply calling the event handlers, and without mimic user interactions on the application. We believe it is mandatory to replay instantiated GUI tests on the application in order to comply with the idea of system testing. For example, timing problems can only be detected when executing the GUI test on the application itself. E.g., the replayer tries to execute an event on a window, but this window is not yet displayed.

The replayer assigns values, e.g., a string value to a text box, by reflection and memory-mapped files. In principle, this may violate an invariant of the application. For example, it may not be allowed to access a certain text box, since the text box is currently disabled. In our approach we use the EFG and its corresponding GUI structure to guess that a widget is accessible. However, since the EFG represents an approximation, it cannot be guaranteed that a widget is actually accessible. A possible alternative is to add annotations to the source code, stating that a value to a widget may only be assigned under specific conditions.

6 Related Work

In [13], Symbolic Java PathFinder is used to generate test cases. The symbolic execution is performed on unit level and combines concrete execution on system level. The use of Pex on a parameterized GUI test can be seen as symbolic execution on unit level. However, in our approach concrete execution on system level takes place when replaying instantiated GUI tests. Further, testing on system level eliminates the problem of executing infeasible sequences [7].

The approach presented in [5] generates test cases for GUI applications using symbolic execution. Our work differs in three main aspects: First, we incorporate a black-box model (an Event Flow Graph) of the GUI in order to select event sequences that are actually executable on the GUI. Second, we generate parameterized GUI tests which can also be used with other techniques than symbolic execution. Third, our approach is able to replay instantiated GUI tests in a black-box fashion on the application.

The work in [8] is related to our work on an abstract level in that it combines black-box and white-box testing. Concretely, however, the underlying technical issues to be solved are incomparable due to the different settings (unit testing vs. system testing, method calls vs. event handlers).

The focus in [1] (with shared co-authors) is to generate test sequences that are at the same time executable and justifiably relevant. Random values for widgets are used, as opposed to generated values as in this paper. We have to use different sets of benchmarks for the experiments, corresponding to the different programming environments (Java vs. C#). The migration of the work in [1] to C# is in progress.

7 Conclusion and Future Work

In this paper we have proposed a novel approach to the generation of GUI tests, implemented in a new tool called *Gazoo*. Gazoo selects event sequences from the EFG of an application and generates a set of *Parameterized GUI tests*. Then, Gazoo applies Pex in order to instantiate parameterized GUI tests. Finally, Gazoo replays instantiated GUI tests on the application. In the terminology of the black-box/white-box dichotomy, Gazoo starts with a black-box approach (using the EFG in order to select executable test sequences), then moves on to a white-box approach (in order to generate parameterized GUI tests and instantiate them using Pex), and finally goes back to the black-box approach (using a replayer in order to execute the (instantiated) GUI tests on the application). As shown in the paper, we needed to overcome a number of non-trivial technical hurdles in order to establish the appropriate interface between the black-box approach and the white-box approach.

The scope of this paper was to show that our approach can achieve high code coverage. Usually one expects that high code coverage translates to high bug detection rate. For future work, we need to evaluate that this holds true in our setting. This evaluation requires its own series of experiments where one applies statistical methods to fault-seeded versions of AUTs, following, e.g., [11, 18].

Our work opens an interesting perspective for future research because the general scheme behind our approach goes well beyond a specific tool, here Gazoo. We need to explore different alternatives such as, e.g., [2, 16] and, e.g., [12, 19] for going back and forth between the black-box approach and the white-box approach in the sense described above.

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.

References

1. S. Arlt, A. Podelski, C. Bertolini, M. Schäf, I. Banerjee, and A. M. Memon. Lightweight Static Analysis for GUI Testing. In *ISSRE*, 2012.
2. F. Belli. Finite-State Testing and Analysis of Graphical User Interfaces. In *ISSRE*, pages 34–43, 2001.
3. J. de Halleux and N. Tillmann. Parameterized Unit Testing with Pex. In *TAP*, pages 171–181, 2008.
4. G. Fraser and A. Zeller. Generating parameterized unit tests. In *ISSTA*, pages 364–374, 2011.
5. 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.
6. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
7. F. Gross, G. Fraser, and A. Zeller. EXSYST: Search-based GUI testing. In *ICSE*, pages 1423–1426, 2012.
8. N. Kicillof, W. Grieskamp, N. Tillmann, and V. A. Braberman. Achieving both model and code coverage with automated gray-box testing. In *A-MOST*, pages 1–11, 2007.
9. A. M. Memon. An event-flow model of GUI-based applications for testing. *Softw. Test., Verif. Reliab.*, 17(3):137–157, 2007.
10. A. M. Memon, I. Banerjee, and A. Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *WCRE*, pages 260–269, 2003.
11. A. M. Memon, I. Banerjee, and A. Nagarajan. What Test Oracle Should I Use for Effective GUI Testing? In *ASE*, pages 164–173, 2003.
12. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *ICSE*, pages 75–84, 2007.
13. C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–26, 2008.
14. N. Tillmann and W. Schulte. Parameterized unit tests. In *ESEC/SIGSOFT FSE*, pages 253–262, 2005.
15. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java PathFinder. In *ISSTA*, pages 97–107, 2004.
16. L. J. White, H. Almezen, and N. Alzeidi. User-Based Testing of GUI Sequences and Their Interactions. In *ISSRE*, pages 54–65, 2001.
17. 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.
18. X. Yuan and A. M. Memon. Using GUI Run-Time State as Feedback to Generate Test Cases. In *ICSE*, pages 396–405, 2007.
19. S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, 2011.