Using Dynamic Symbolic Execution to Generate Inputs in Search-Based GUI Testing

Kevin Salvesen∗, Juan P. Galeotti∗, Florian Gross∗, Gordon Fraser† and Andreas Zeller∗
∗Computer Science, Saarland University, Germany
†Dep. of Computer Science, University of Sheffield, UK

Abstract—Search-based testing has been successfully applied to generate complex sequences of events for graphical user interfaces (GUIs), but typically relies on simple heuristics or random values for data widgets like text boxes. This may greatly reduce the effectiveness of test generation for applications which expect specific input values to be entered in their GUI by users. Generating such specific input values is one of the virtues of dynamic symbolic execution (DSE), but DSE is less suitable to generate sequences of events. Therefore, this paper describes a hybrid approach that uses search-based testing to generate sequences of events, and DSE to build input data for text boxes. This is achieved by replacing standard widgets in a system under test with symbolic ones, allowing us to execute GUIs symbolically. In this paper, we demonstrate an extension of the search-based GUI testing tool EXSYST, which uses DSE to successfully increase the obtained code coverage on two case study applications.

I. INTRODUCTION

The most common system-level interface of software applications is their graphical user interface (GUI) [1]. It is estimated that 45% to 60% of all software code is used for implementing GUIs and that 50% of total development time is spent on GUIs [2]. Compared to other types of code, GUI-based software has a reputation of being particularly hard to test [3]. Testing GUI-based software manually is labour intensive, as even small applications can have GUIs with many screens and widgets [4].

To support GUI testing, automation is desired [5]. The simplest automated GUI testing approaches are random [6] or script-based testing [7], but more advanced techniques relying on model-based testing [8] or search-based testing [9] have been proposed. Most of these tools focus entirely on generating complex sequences of GUI events that try to cover an application’s GUI as much as possible, and ignore the problem of generating input values for data widgets like text boxes. While using random input values (as most tools do) can still lead to high coverage on some applications, on those that expect specific input for their data widgets the achieved coverage is usually much lower.

For example, consider the WorkoutGenerator application, shown in Figure 1, which generates a workout plan for a user depending on attributes like age, size, and sex. Even a state of the art GUI testing tool like EXSYST [9] fails to cover large parts of its code. Listing 1, showing an excerpt of WorkoutGenerator’s code, reveals why: There are branches using seemingly simple logical conditions — but these conditions depend on the values users enter for their age, height, and weight in text boxes. Using random strings (which may not even be numeric) for these text boxes thus means that most of the application’s behaviour is never tested.

To solve this issue, we extend the search-based automated GUI test generator EXSYST, such that it applies dynamic symbolic execution to generate inputs for text boxes.
symbolic execution (DSE) to intelligently generate inputs for data widgets.

II. BACKGROUND

A. Search-based GUI Test Generation

EXSYST [9] is an automated test generator for GUI based applications. Its goal is to create a set of GUI test cases that achieve high code coverage while maintaining the test suite size as small as possible. The implementation of EXSYST is heavily based on the EvoSuite [10] unit test generator. EvoSuite applies a genetic algorithm to evolve a population of whole test suites towards high code coverage (among other criteria). Chromosomes and search-operators in EXSYST are adapted from EvoSuite. Each chromosome represents one GUI test suite composed of a set of test cases. As shown in Figure 3, each test case is a sequence of GUI actions, such as clicking on a button or entering data on a text box.

As GUI elements could be enabled or disabled due to the execution of GUI actions, EXSYST maintains a model of the application. This model represents all the known application behaviour explored so far by EXSYST. Nodes in the model represent GUI states (a state can be defined as a set of all the components available in a GUI), and transitions represent GUI actions and their resulting GUI state. For example, if the Clear button in the current state is disabled, by entering a value in the Age text box will change the current state to another in which the button is now enabled.

Listing 1. A Test case produced by EXSYST with DSE for the Tickets GUI application

```
ButtonClick on Button[“Button1”]
EnterText[“m07s=k”] on TextField #0
ButtonClick on Button[“Button2”]

ToggleButton on Button[“Label1”]

EnterText[“otf9K+Y6;YM”] on TextField #0
ButtonClick on Button[“Button2”]

ComboBoxSelect[0.0] on javax.swing.JComboBox[“[Business, Student]”] #0
ComboBoxSelect[0.3333333333333333] on javax.swing.JComboBox[“[Business, Student]”] #0
ComboBoxSelect[0.6666666666666666] on javax.swing.JComboBox[“[Business, Student]”] #0
ComboBoxSelect[0.9999999999999999] on javax.swing.JComboBox[“[Business, Student]”] #0

ComboBoxSelect[0.0] on javax.swing.JComboBox[“[Business, Student]”] #0
ComboBoxSelect[0.3333333333333333] on javax.swing.JComboBox[“[Business, Student]”] #0
ComboBoxSelect[0.6666666666666666] on javax.swing.JComboBox[“[Business, Student]”] #0
ComboBoxSelect[0.9999999999999999] on javax.swing.JComboBox[“[Business, Student]”] #0

ComboBoxSelect[0.0] on javax.swing.JComboBox[“[Business, Student]”] #0
ComboBoxSelect[0.3333333333333333] on javax.swing.JComboBox[“[Business, Student]”] #0
ComboBoxSelect[0.6666666666666666] on javax.swing.JComboBox[“[Business, Student]”] #0
ComboBoxSelect[0.9999999999999999] on javax.swing.JComboBox[“[Business, Student]”] #0
```

Fig. 3. A sample EXSYST chromosome representing a GUI test suite composed of three test cases. Each test case is a sequence of GUI actions (here on a very simple program composed of two buttons and one text box).

B. Dynamic Symbolic Execution

Dynamic Symbolic Execution [11], [12] combines symbolic execution with concrete executions of the program using concrete values. During the execution of the program, a symbolic state of the system is maintained. Whenever a change in the program state occurs, the symbolic state is updated accordingly. This combined concrete and symbolic execution is usually called dynamic symbolic execution.

After the program execution is finished, a path condition is collected (representing under which conditions inputs will traverse the same program path). In turn, the negated path condition is fed to a constraint solver to generate a concrete input that will traverse a different program path.

C. Hybrid Search-based Test Generation and DSE

In previous work [13] we improved the effectiveness of EvoSuite by applying dynamic symbolic execution. All primitive values declared in a test case (i.e. String, int and float values) are marked as symbolic, and the path conditions for these symbolic variables can be used to generate new concrete values executing different paths. To reduce the overhead caused by the dynamic symbolic execution, it can be applied adaptively whenever a mutation of a primitive value (String, int or float) leads to a change in the fitness value of the test suite, and the success of each application of DSE can be used to influence the frequency of applying DSE.

III. DYNAMIC SYMBOLIC EXECUTION AT THE GUI LEVEL

Our prototype profits considerably from the DSE extension of EvoSuite. In particular, we were able to apply almost straightforwardly the mechanism for gathering path conditions, as well as the search-based constraint solver integrated within EvoSuite [14]. The current implementation of EXSYST applies DSE on every new generation of the genetic algorithm. In other words, EXSYST does not implement yet the heuristics presented in [14] to adaptively apply DSE.

In the remaining of this section we describe some implementation details of DSE in EXSYST.

A. Symbolic Widgets

Each GUI action is specified in a test case as a triplet (eventId, parameterList, widgetId). For example, the GUI-action

```
EnterText[“m07s=k”] on TextField #0
```

shown in Figure 3 has EnterText as eventId, the parameter list is the singleton list of the String value “m07s=k” and the widgetId value is TextField #0. Some GUI-actions have an empty parameter list (like all ButtonClick events).

Only primitive values (i.e. String, int and float) are used as parameters for GUI-actions. During symbolic
execution, EXSYST marks all these primitive values as symbolic values, and conditions upon them are collected during DSE. Each event of a GUI-action corresponds to a method in the actual widget implementation. For example, `EnterText` maps to method `setText(String)` of class `JTextField`, and `ComboBoxSelect` maps to method `setSelectedIndex(int)` of class `JComboBox`.

Widget methods can be very complex (for example, the execution of `setText` can span up to five different classes involving data structures and native code), making the application of symbolic techniques generally difficult. Due to this, GUI testing frameworks (like BARAD [15] and GAZOO [16]) introduce the concept of symbolic widgets, which are simplified versions of the GUI widgets. In contrast to standard GUI widgets, symbolic widgets focus solely on functionality (while regular widgets also deal with presentation and performance). For the case of the `JTextField` widget, the symbolic counterpart of that widget stores the current text value as a simple `String` value, that can be accessed with trivial getters and setters.

EXSYST automatically replaces each occurrence of a concrete widget with its symbolic counterpart before DSE is applied. As symbolic widgets are semantic-preserving versions of the concrete widgets, the benefits of symbolic widgets is twofold: first DSE is more efficient since path conditions are shorter. Secondly, the exploration naturally avoids the concrete widget implementation, allowing EXSYST to focus on the application code.

B. Symbolic execution of GUI threads

In order to symbolically execute the target Java program and gather the path condition, EXSYST instruments each bytecode instruction by adding a callback. During the callback execution, the symbolic state is updated accordingly before the actual concrete instruction is executed (similarly the path condition is updated if the bytecode instruction is a branch instruction).

Java Swing handles events using a special thread called the event dispatch thread [17]. This thread is also responsible for handling other events than those executed by our test cases. For example, the `RepaintManager` class of the Swing framework (which is responsible for redrawing an application’s GUI) frequently makes calls that access a GUI application under test.

As all threads share the same instrumented code, callbacks from the event dispatch thread could leave the symbolic state inconsistent (or even lead to a crash during within dynamic symbolic execution framework). EXSYST checks if the current thread corresponds to the dynamic symbolic execution of a test case chromosome. This check is done by inspecting the stack trace of the current thread. If so, the update of the symbolic state is performed. Otherwise it is discarded, keeping the symbolic state consistent.

IV. CASE STUDIES

We executed EXSYST on two case studies, which were previously used to evaluate the BARAD testing tool [15].

<table>
<thead>
<tr>
<th>Case Study</th>
<th>LOC</th>
<th>EXSYST</th>
<th>EXSYST +DSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tickets</td>
<td>272</td>
<td>36.96%</td>
<td>70.89%</td>
</tr>
<tr>
<td>WorkoutGen.</td>
<td>401</td>
<td>63.13%</td>
<td>88.13%</td>
</tr>
</tbody>
</table>

While the initial case studies look promising, our prototype is still under development. Some challenges we will face in the near future include:

More symbolic widget support: Our current prototype is limited to a subset of widgets (mostly, those that are extensions of `JTextField`). We would like to replace more widgets with symbolic counterparts, such as `table widgets`. This is specially challenging since Java Swing table widgets are defined with the intent that the user should redefine some of their components, such as the model holding the tables data (by subclassing `DefaultTableModel`) or the GUI widgets responsible for editing a cell in the table (by replacing `DefaultCellEditor`). As an example, the simple Spreadsheet application [9] extends its table’s cell editors so that they behave differently when a formula is entered into the table.

Applying an Adaptive approach for DSE: In previous work [14], an adaptive approach for deciding whenever DSE could be more beneficial was introduced. In the experiments we have performed for this work, DSE was applied on every single generation of the genetic algorithm. Although the
current implementation of EXSYST also allows to configure how often DSE can be applied, deciding which rate is optimal could be complex. Due to this, we would like to extend that adaptive approach for GUI applications.

VI. RELATED WORK

BARAD [15] is a GUI testing framework for applications written in Java with the Standard Widget Toolkit (SWT). GAZOO [16] is a fully automated GUI testing tool for .NET applications. Both BARAD and GAZOO rely on DSE to find input data for event sequences but, unlike the search-based approach used in EXSYST, GUI-sequences are exhaustively executed up to a certain limit. COLLIDER [18] targets Android applications. It uses concolic execution to build symbolic summaries of the GUI’s event handlers, but requires a UI model to derive proper sequences of GUI-actions. In contrast, EXSYST requires no previously inferred UI model.

Approaches combining SBST and DSE are not new. Inkum-sah and Xie [19] proposed a combination of SBST with DSE by hooking together an evolutionary testing tool and a DSE tool, such that either DSE optimizes the results of the SBST tool, or vice versa. Baars et al. [20] introduced the idea of symbolic search-based testing, where the fitness function of the GA counts the number different symbolic paths to the target. Malburg and Fraser [21] applied DSE as a single mutation operator for improving unit-level test generation. Harman et al. [22] presented a mutation-based test data generation approach combining DSE and SBST. Unlike EXSYST, none of the above can perform GUI-testing.

VII. CONCLUSIONS

We have extended the EXSYST search-based test generator for GUI applications such that it applies DSE to optimize the values for inputs such as text boxes. While we are not the first to use some form of symbolic execution to generate GUI test cases, to the best of our knowledge we are the first to combine it with a search-based GUI testing technique.

Our initial results show that our hybrid method can provide great benefits when testing GUI applications.

ACKNOWLEDGMENT

The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement nr. [290914] and EU FP7 grant 295261 (MEALS).

REFERENCES


