Event Listener Analysis and Symbolic Execution for Testing GUI Applications

Svetoslav Ganov, Chip Killmar, Sarfraz Khurshid, Dewayne E. Perry
The University of Texas at Austin
Austin TX, USA
{svetoslavganov}@mail.utexas.edu, {khurshid, perry}@ece.utexas.edu

Abstract. Graphical User Interfaces (GUIs) are composed of virtual objects, widgets, which respond to events triggered by user actions. Therefore, test inputs for GUIs are event sequences that mimic user interaction. The nature of these sequences and the values for certain widgets, such as textboxes, causes a two-dimensional combinatorial explosion. In this paper we present Barad, a GUI testing framework that uniformly addresses event-flow and data-flow in GUI applications generating tests in the form of event sequences and data inputs. Barad tackles the two-dimensional combinatorial explosion by pruning regions of the event and data input space. For event sequence generation we consider only events with registered event listeners, thus pruning regions of the event input space. We introduce symbolic widgets which allow us to obtain an executable symbolic version of the GUI. By symbolically executing the chain of listeners registered for the events in a generated event sequence we obtain data inputs, thus pruning regions in the data input space. Barad generates fewer tests and improves branch and statement coverage compared to traditional GUI testing techniques.

Keywords: GUI testing, symbolic execution, test input generation

1 Introduction

A Graphical User Interface (GUI) provides a convenient way to interact with the computer. GUIs consist of virtual objects (widgets) that are intuitive to use, for example, buttons, edit boxes, etc. In contrast to console applications where there is only one point of interaction (the command line), GUIs provide multiple points (the GUI widgets) each of which can have different states.

A classic challenge in GUI testing is how to select a feasible number of event sequences, given the combinatorial explosion due to arbitrary event interleavings. Consider testing a GUI with five buttons, where any sequence of button clicks is a valid input. Exhaustive testing without repetition requires trying all 120 combinations since triggering one event before another may cause execution of different code paths.

An orthogonal challenge is how to select values for *data widgets*, i.e., GUI widgets that accept user input, such as textboxes, edit-boxes and combo-boxes, and can have an extremely large space of possible inputs. Consider testing a GUI with one text-box that takes a ten character string as an input. Exhaustive testing requires 10^{26} possible input strings (we limit each character to be alphabetical in lower-case).

Automation of GUI testing has traditionally focused on minimizing event sequences [9] [10] [12] [18] [21]. Data widgets have either been abstracted away by not considering GUI behaviors dependent on data values, generated at random, or selected from a small manually constructed set of values. As a consequence, data dependent behaviors are inadequately tested. Consider generating a string value that is necessary for satisfying an *if*-condition. Random selection is unlikely to generate the desired value. Manual selection requires tedious code inspection. A specification-based (black-box) approach may find this “special” value, however it would require detailed specifications, which often are not provided.

In this paper we present Barad, a novel GUI testing framework for checking GUI applications written in Java with the Standard Widget Toolkit (SWT) [20]. Barad generates event sequences and data inputs providing a systematic approach that uniformly addresses event-flow and data-flow for white-box testing of GUI applications. We detect *event listeners* – instances that register for and respond to events in the GUI. This allows us to consider only events with registered listeners during event sequence generation, thus pruning regions of the event input space. We symbolically execute the chain of listeners registered for the events in a generated event sequence. This allows us to obtain data inputs for the fields of GUI widgets, thus pruning regions in the data input space. Barad is fully automatic, performing bytecode instrumentation, test generation, symbolic execution, and test execution. While our current implementation handles only GUIs written with the SWT library, our approach is generic and can be successfully applied to other Java GUI libraries.

To scale symbolic execution [7] [8] [15] [16] [17] for GUI applications, we introduce symbolic widgets which allow us to perform symbolic manipulation of standard GUI widgets and obtain an executable symbolic version of the GUI. Widget implementations have three concerns: (1) functionality; (2) visualization; and (3) performance. Symbolic widgets focus on functionality, abstracting away the other two concerns. The benefit of this approach is that it enables (1) efficient and systematic dynamic analysis of GUI applications, and (2) generation of inputs for data widgets.

In our previous publication [5] we introduced symbolic analysis of GUI event listeners in isolation for obtaining data inputs without considering event sequence generation and analysis of event listeners in the context of the GUI application (i.e. interactions between event listeners). In this paper we present a novel approach for event sequence generation and symbolic analysis of the GUI application.

We make the following contributions:

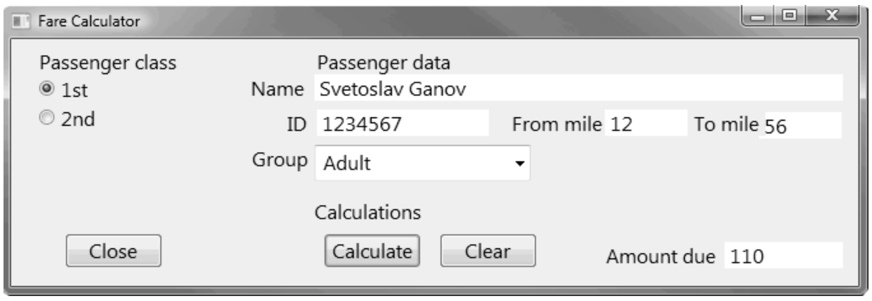
- **Symbolic analysis of GUI applications.** We introduce the abstraction of symbolic widgets that enables efficient and systematic dynamic analysis of GUI applications.
- **Event sequence generation.** We present a novel test generation approach and consider only events with registered event listeners, thus pruning regions of the event input space.
- **Data input generation.** By symbolically executing the chain of listeners registered for the events in a generated event sequence we obtain data inputs, thus pruning regions in the data input space.
- **Implementation.** Barad is fully automatic, performing Java bytecode instrumentation, test generation, symbolic execution, and test execution.
- **Evaluation.** We evaluate our approach on non-trivial GUI subjects and compare it to traditional GUI testing techniques. Barad generates fewer tests and achieves higher statement and branch coverage.

2 Example

In this section we provide an example how our approach uniformly handles event-flow and data-flow in GUI applications and compare it to conventional GUI testing techniques.

2.1 Fare Calculator

The GUI presented in Figure 1 is an application (313 lines of code) that we developed. It calculates the amount due for a train ticket. A user must provide a passenger class, name, ID, passenger group, and begin and end points. Passenger groups are *Senior*, *Adult*, *Student*, and *Child*.


Fare Calculator

Passenger class: [✓ 1st] 2nd

Passenger data

Name: Svetoslav Ganov

ID: 1234567 From mile: 12 To mile: 56

Group: Adult

Calculations

Amount due: 110

Fig. 1. Screenshot of the Fare Calculator.

Each passenger class has its own coefficient that is used during the calculation. Each group has a different base price depending on the distance to be traveled, which is the difference between *From mile* and *To mile*. This application has three event listeners registered for the click events in buttons *Close*, *Calculate*, and *Clear*, respectively. The calculation logic has 22 branches with conditional statements nested three levels deep. The execution of a particular branch depends on the user input both in the form of data and event sequence.

2.2 Input space

The Fare Calculator consists of two radio buttons, five textboxes, three buttons, and one combo–a total of eleven GUI widgets. Therefore, the number of event sequences with only one event per widget and one value per data widget is slightly less than 4,000,000 (11!). Furthermore, just the input for the ID field of the Fare Calculator, a sequence of ten numeric characters, causes a factor of 10,000,000,000 (10^{10}) increase in the test suite size. Hence, due to the two-dimensional combinatorial explosion in GUI inputs, exhaustive testing of even as simple GUI as the Fare Calculator is unrealistic. Clearly, a systematic approach that prunes regions of the event and data input space is required.

2.3 Test results

We tested the Fare Calculator with Barad. The process was completed fully automatically. Our results are shown in Table 1.

Table 1. Test results with enabled symbolic analysis.

Tests	Branch coverage, %	Statement coverage, %	Time, sec
69	100	100	13.02

The first column presents the total number of tests. The second and the third columns present the branch and statement coverage, respectively. Column four contains the execution time which includes instrumentation, test generation, symbolic execution, and test execution. Barad uses Emma [4] to determine code coverage. Branch coverage was obtained by manual inspection of the code coverage report.

We interpret our results as follows. The application has three event listeners registered for the events of clicking each of the buttons. Hence, during event sequence generation we consider only these three events resulting in six tests with length three without repetition. Each test case was executed on the symbolic version of the GUI and for some tests sets of input values were obtained. Test cases, symbolic execution of which generated sets of input values, were prefixed with events to populate each set of values, thus producing a new test case for each input set. The full branch and statement coverage is due to data values obtained by systematic exploration of all feasible paths during symbolic execution.

Conventional GUI testing techniques [9] [10] [12] [18] [22] exhaustively generate event sequences up to a given bound and adopt a specification based approach to populate inputs—selecting from a predefined set of values. We disabled the symbolic and event listener analysis in Barad to simulate conventional GUI testing. We limited the length of event sequences to be equal to the length of sequences generated by our approach before prefixing with events for data input population. The input values for data widgets were chosen in a widget specific manner as follows: for the textboxes a choice from the set {*-1*, *0*, *1*, *Test*, *ThisIsAVeryLongStringValue*, the empty string} was made; for the combo a choice from the set of possible values, namely {*Senior*, *Adult*, *Student*, *Child*} was made. Results of this analysis are presented in Table 2. The first column presents the total number of tests. The second and the third columns present the branch and statement coverage, respectively. Column four contains the execution time which includes test generation and test execution.

Table 2. Test results with disabled symbolic analysis.

Tests	Branch coverage, %	Statement coverage, %	Time, sec
1152	23	87	142.45

2.4 Comparison

Results show that for the Fare Calculator our approach generated more than an order of magnitude fewer tests compared to a traditional approach, while achieving significantly higher branch coverage. The longest event sequence generated by our technique has length eight and consists of the following events: (1) selecting a *Passenger class*; (2) populating the *Name* field; (3) populating the *ID* field; (4) populating the *From mile* field; (5) populating the *To mile* field; (6) selecting the *Calculate* button; (7) selecting the *Clear* button; (8) selecting the *Close* button; Note that our approach generated the minimal set of event sequences with length eight to achieve full path coverage. In contrast, to generate a test case with this length and achieve the same coverage results the traditional approach requires generation of all event sequences with length eight without repetition. Considering the very limited input specifications, this results in 7.6×10^{13} test cases.

3 Background

This section provides the reader with some background about the technique of symbolic execution. It also presents the traditional GUI testing approaches and the GUI model we adopt.

3.1 Symbolic Execution

The main idea behind symbolic execution is to use *symbolic values*, instead of actual data, as input values, and to represent the values of program variables as symbolic expressions. As a result, the output values computed by a program are expressed as a function of the input symbolic values.

The *state* of a symbolically executed program includes the (symbolic) values of program variables, a *path condition* (PC), and a program counter. The path condition is a (quantifier-free) Boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path. The program counter defines the next statement to be executed. A *symbolic execution tree* characterizes the execution paths followed during the symbolic execution of a program. The nodes represent program states and the arcs represent transitions between states.

Consider the code fragment in Figure 2, which swaps the values of integer variables x and y, when x is greater than y. The figure also shows the corresponding symbolic execution tree. Initially, PC is *true* and x and y have symbolic values X and Y, respectively. At each branch point, PC is updated with assumptions about the inputs, in order to choose between alternative paths. For example, after the execution of the first statement, both then and else alternatives of the *if*-statement are possible, and PC is updated accordingly.

```
1 int x, y;
2 if (x > y) {
3   x = x + y;
4   y = x - y;
5   x = x - y;
6   if (x - y > 0)
7    assert(false);
8  }
9 }
```

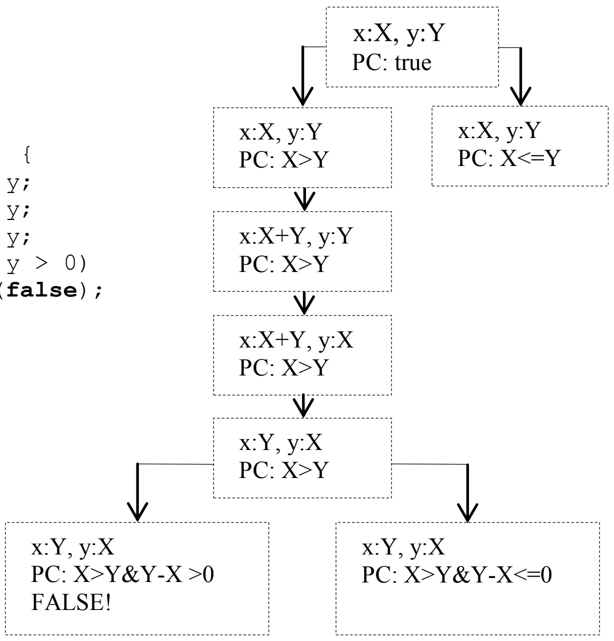


Fig. 2. Code that swaps two integers and its symbolic execution tree where transitions are labeled with program control points.

If the path condition becomes *false*, i.e., there is no set of inputs that satisfy it, this means that the symbolic state is not reachable, and symbolic execution does not continue for that path. For example, statement (7) is unreachable.

3.2 GUI testing approaches

Since contemporary software extensively uses GUIs to interact with users, verifying GUI's reliability becomes important. There are two approaches to building GUIs and these two approaches affect how testing can be performed.

The first approach is to keep the GUI light weight and move computation into the background. In such cases the GUI could be considered as a "skin" for the software. Since the main portion of the application code is not in the GUI, it may be tested using conventional software testing techniques. However, such an approach places architectural limitations on system designers.

The second approach is to merge the GUI and its computations. The most common way of testing such GUIs is by using tools that record and replay event sequences. This is laborious and time consuming. Another technique for checking GUI's correctness is by using tools for automatic test generation, execution, and assessment as the one presented in this paper or the ones described in [9] [12] .

3.3 GUI model

We take a standard view of a GUI. Let $W = \{w_1, w_2, ... w_n\}$ be the set of GUI widgets. Examples of widgets are *Button, Combo, Label,* etc. Each widget has a set of properties. Let $P = \{p_1, p_2, ... p_m\}$ be the set of widget properties. Examples of properties are *enabled*, *text*, *visible*, *selection*, etc. Each property has a set of values. Let $V = \{v_1, v_2, ... v_p\}$ be the set of property values. Examples of values are *true*, *false*, etc. A GUI is a triple (W, ρ, ν) that consists of a set of widgets, a mapping $\rho : W \rightarrow 2^P$ from widgets to properties, and a mapping $\nu : P \rightarrow 2^V$ from properties to values.

Let E be the set of all events accepted by the GUI. Each GUI widget w accepts as input a set of user events E_w triggered by user actions which is a subset of E. Examples of events are *clicks*, *mouse moves,* etc.

$$\forall w \in W \mid \exists E_w \subseteq E : accept(w, E_w) \quad \textbf{(1)}$$

Let L be the set of all event listeners in the GUI. Each GUI widget w has zero or more event listeners L_w registered for events performed on the widget which is a subset of L. Each listener l is registered for a set of events E_l which is a subset of all events E_w accepted by the widget. Examples of listeners are *selection listener*, *modification listener*, etc.

$$\forall w \in W \mid \exists L_w \subseteq L \wedge \forall l \in L_w \mid \exists E_l \subseteq E_w \wedge \forall e \in E_l \mid registered(l, e) \quad \textbf{(2)}$$

Since a user interacts with the GUI through events, a GUI test case t from the set T of GUI test cases is an event sequence.

$$\forall t \in T : t = < e_1, e_2, ..., e_p > \quad \textbf{(3)}$$

4 Barad

This section presents Barad, our GUI testing framework. We present the techniques for addressing event-flow and data-flow in GUI applications and our approaches for pruning regions in the event and data input space. We also provide details about the adopted abstractions.

The process of GUI testing performed by Barad is shown on Figure 3. To enable symbolic execution, Barad instruments the bytecode of the tested GUI application replacing concrete entities (widgets, strings, primitives, library classes) with their corresponding symbolic equivalents (symbolic widgets, symbolic strings, symbolic integers etc.) provided by Barad's symbolic library. The bytecode instrumentation is implemented with the ASM library [1]. As a result of the instrumentation phase an executable symbolic version of the GUI is generated. Next, a symbolic analysis of the instrumented version is performed.

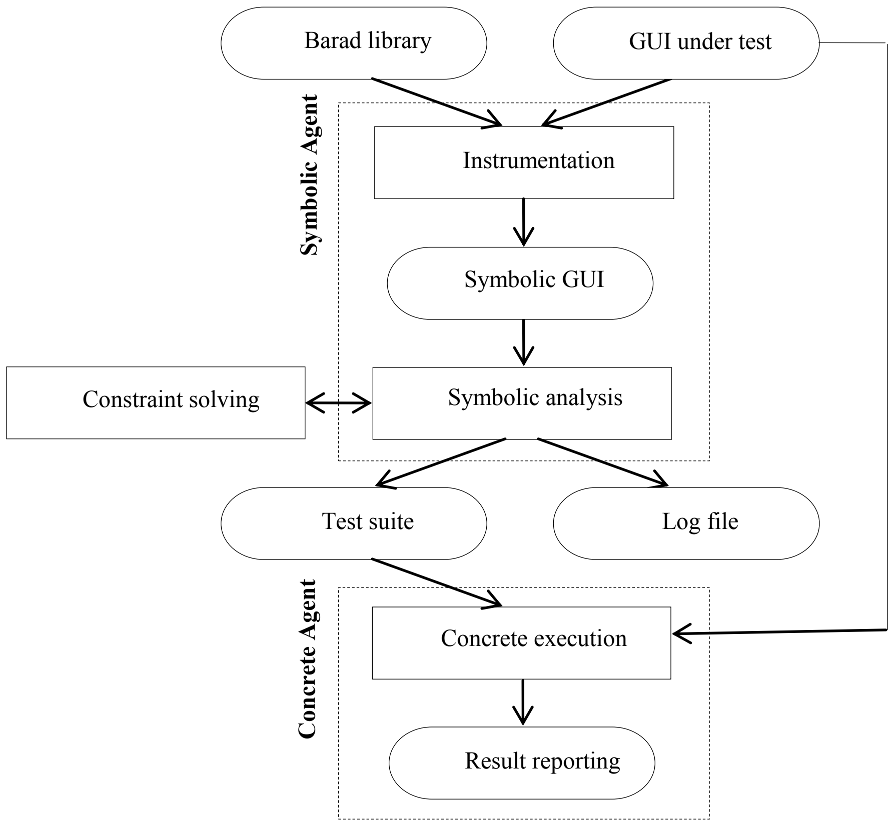


Fig. 3. GUI testing process in Barad.

During this process event listeners are detected, tests in the form of sequences of events with registered listeners are generated, and then symbolically executed—all paths are systematically explored and their feasibility evaluated by constraint solving.

As a result of this process a log file and a test suite are generated. The test suite consists of event sequences and concrete inputs. Finally, the test suite is executed on the concrete version of the application and a coverage report is generated.

4.1 Event-flow

To address event-flow in GUI applications we adopt a strategy of pruning regions in the event input space by not considering events for which there is no registered event listener. Since an event listener contains computational logic performed upon a certain event, the lack of a listener for an event renders the event to have no effect on the GUI.

However, such an approach might prevent the execution of a given program path. Consider a simple GUI with one textbox, one button, and a single event listener for the event of pressing the button. Now assume the event listener code has a conditional statement which depends on the value of the textbox.

Since there is no listener for the event of populating the text box and we consider only events with registered listeners, an event for populating the textbox will not be included in any test case. This leads to inadequate testing because of a failure to cover all program paths in the listener. Hence, adopting a strategy for considering only events with registered listeners requires a mechanism for detecting if the code in the event listener in our example depends on the value of the textbox. To determine such a dependency we perform symbolic analysis of the event listener code and generate values for the textbox that would ideally achieve full path coverage of the event listener. Let us assume that we have identified two values "A" and "B" for the textbox which would force the execution to follow different paths at the conditional statement in the listener. In such a case if we generate two tests one including the event for populating "A" in the textbox followed by pressing the button and the other including the event for populating "B" in the textbox followed by pressing the button we will achieve full path coverage of the event listener. Therefore, employing symbolic execution for identifying such dependencies allows us to safely consider only events with registered event listeners. We generate events that populate data widgets, for which no listener exists only in case we have identified values that would force visiting unexplored program paths. More details about our test generation approach are presented in Section 4.4.

To illustrate the reduction in the event input space by considering only events with registered event listeners consider the Fare Calculator from Section 2. The GUI consists of eleven widgets and three event listeners. Considering only one event per widget (some widgets accept more than one event) results in 165 event sequences with length three while considering only events with registered listeners results in only six events sequences with length three.

4.2 Data-flow

To address data-flow in GUI applications we utilize symbolic execution to obtain inputs for data widgets. We execute symbolically the chain of event listeners registered for the events in a test case. This is achieved by executing each test case on a symbolic version of the application.

In order to obtain a symbolic version of the application, thus enabling symbolic execution of GUIs, we introduce the abstraction of symbolic widgets. Each GUI widget has a symbolic counterpart that has the same fields and provides the same methods, which however represent and operate on symbolic data, respectively. For example, *org.eclipse.swt.widgets.Text* is mapped to a *barad.symboliclibrary.ui.widgets.SymbolicText* and the string field *text* of the former is implemented as a symbolic string in the latter. The corresponding *getter* and *setter* methods, for the *text* field of the *SymbolicText* widget, return as a result and receive as a parameter symbolic strings. To enable the integration of symbolic widgets in our framework, we introduce *symbolic events* and *symbolic event listeners*. Similarly to symbolic widgets, these entities are structurally equivalent to their concrete counterparts and operate with symbolic data.

Symbolic widgets could be envisioned as wrappers that relate sets of variables, representing symbolic primitives and strings, to particular instances in the GUI widget hierarchy. Therefore, constraints and operations on symbolic widgets are constraints and operations on symbolic primitives and strings.

However, symbolic widgets have richer semantics than the set of variables they encapsulate, performing specific to the symbolic and event listener analysis functions: (1) Symbolic widgets wrap the variables related to concrete GUI widgets, allowing us to maintain a mapping from symbolic variables to concrete GUI widgets. This mapping identifies which concrete widgets to be populated with values obtained after concretization of symbolic variables; (2) Symbolic widgets are mapped one-to-one with concrete widgets. This guarantees that the symbolic widget hierarchy is isomorphic to the concrete widget hierarchy and tests generated for the symbolic version of the GUI are applicable to its concrete version; (3) Symbolic widgets detect event listeners at run time. Detecting of event listeners is required by our test generation algorithm; (4) Symbolic widgets implement methods which execute registered event listeners, passing as a parameter a symbolic event. These methods are used for execution of the generated tests; (5) Symbolic widgets, similarly to their concrete counterparts, are referenced by the events passed as parameters to the event listeners. This provides a mechanism of accessing properties of symbolic widgets through events instances.

Symbolic widgets abstract away the visualization layer of their concrete replicas. Such an approach has several advantages. (1) We avoid symbolic execution of the GUI library implementation and focus our analysis on the application logic. Our objective is verifying application correctness, rather than proper behavior of the GUI library. (2) We avoid the native calls made by a GUI widget to the operating system to generate a visual representation of the widget. Our focus, during symbolic execution, is on data-flows in GUI applications and the visual representation of these GUIs is irrelevant to our analysis. Hence, we abstract away unnecessary computations.

Currently Barad supports the symbolic widgets, events, and event listeners required for testing the GUI applications presented in this paper. Our framework is an experimental prototype used to evaluate the applicability of our approach. We did not encounter any widget specific issues, which make defining a symbolic widget challenging. We believe that full support for the SWT library as well as other Java GUI libraries is feasible.

4.3 Symbolic GUI model

Our view of the symbolic version of a GUI follows the GUI model we have presented in Section 3.3.

Let $W_s = \{w_{s1}, w_{s2}, \ldots w_{sn}\}$ be the set of symbolic widgets. Each symbolic widget has a set of properties which are symbolic variables $P_s = \{p_{s1}, p_{s2}, \ldots p_{sm}\}$. Each symbolic property has a set of values it can take during its concretization $V_s = \{v_{s1}, v_{s2}, \ldots v_{ps}\}$. A symbolic GUI is a triple (W_s, ρ, ν) that consists of a set of symbolic widgets, a mapping $\rho : W_s \to 2^{P_s}$ from symbolic widgets to symbolic properties, and a mapping $\nu : P_s \to 2^{V_s}$ from symbolic properties to concrete values.

Let E_s be the set of symbolic events. Each symbolic widget w_s accepts as input a set of symbolic events E_{ws}.

$$\forall w_s \in W_s \mid \exists E_{ws} \subseteq E_s : accept(w_s, E_{ws}) \qquad \textbf{(1)}$$

Let L_s be the set of event listeners. Each symbolic widget w_s has zero or more event listeners L_{ws}. Each listener l_s is registered for a set of symbolic events E_{ls}.

$$\forall w_s \in W_s \mid \exists L_{ws} \subseteq L_s \wedge \forall l_s \in L_{ws} \mid \exists E_{ls} \subseteq E_{ws} \wedge \forall e_s \in E_{l_s} \mid registered(l_s, e_s) \qquad \textbf{(2)}$$

4.4 Test generation algorithm

Taking advantage of the symbolic widgets we developed our test generation algorithm shown in Figure 4.

```
1. SymbolicModel.executeEventsWithListeners();
2. eventSequences = TestGenerator.generateTests();
3. for (EventSequence s: eventSequences){
4.   in = SymbolicModel.excecuteListenerSequence(s.listeners());
5.   test.addAll(TestGenerator.appendInputs(in, s);
6. }
```

Fig. 4. Test generation algorithm.

We represent the GUI events with registered event listeners as an *Events with Listeners Graph* (ELG)—a directed graph with nodes representing events with registered listeners and edges. The existence of an edge from event *e1* to event *e2* means an execution of event *e2* can be performed immediately after the execution of event *e1*. For example, if event *e1* opens a new *form* (GUI window) every event in that form strictly succeeds *e1*. Every time a new event with registered listener is identified a new node is added to the graph.

Since events with registered listeners are detected at runtime by symbolic widgets (intercepting event listener registration calls) and these events can open other forms, all events with registered listeners should be executed at least once (line 1) to build a complete ELG. Such an approach enables handling of multiple GUI windows. Once an ELG has been created we generate test cases performing graph traversals. Our test generation algorithm generates exhaustively test cases in the form of event sequences up to a given bound without repetition (line 2).

We obtain data inputs by symbolically executing the sequence of listeners registered for the events in a test case (line 3-6). Doing so, we capture data dependencies between the event listeners and potentially identify sets of input values for the data widgets in the GUI (line 4). For each such set (if such sets exist) a test case is created by concatenating events for populating data widgets with the values from the set and the events of the test case (line 5).

To illustrate our test generation algorithm, recall the Fare Calculator from Section 2. The algorithm proceeds as follows. Once the symbolic version of the GUI is launched the ELG is constructed by executing every event with registered listener in the GUI (line 1). As a result from this step all three events with registered listeners (for clicking the three buttons) *e*1*, e*2*,* and *e*3 are identified and used for construction of six event sequences (line 2). The listeners corresponding to these events are symbolically executed (line 4). Without loss of generality, consider the event sequence (*e*1, *e*2, *e*3) symbolically executing the listeners of which generated twenty two sets *S* of five inputs values *v*1 – *v*5 each:

$$(e_1, e_2, e_3) \rightarrow \{S_1\{v_1,.., v_5\}, S_2\{v_1,.., v_5\},...., S_{22}\{v_1,.., v_5\}\} \quad \textbf{(1)}$$

Each input set transitions the GUI to such a state that executing the sequence (*e*1, *e*2, and *e*3) will force visiting of a different program path. Our algorithm constructs a separate test case for each set of values by concatenating the event sequence required to populate these values with the test event sequence (line 5). The generated test cases, where *e*(*x, y*) is the event required for populating the value *x* from value set y, look as follows:

$$\mathrm{e}(\mathrm{v}_1, \mathrm{S}_1), \mathrm{e}(\mathrm{v}_2, \mathrm{S}_1), \ldots, \mathrm{e}(\mathrm{v}_5, \mathrm{S}_1), \mathrm{e}_1, \mathrm{e}_2, \mathrm{e}_3; \quad \textbf{(2)}$$

$$\ldots$$

$$\mathrm{e}(\mathrm{v}_1, \mathrm{S}_{22}), \mathrm{e}(\mathrm{v}_2, \mathrm{S}_{22}), \ldots, \mathrm{e}(\mathrm{v}_5, \mathrm{S}_{22}), \mathrm{e}_1, \mathrm{e}_2, \mathrm{e}_3; \quad \textbf{(3)}$$

4.5 Symbolic widget example

To provide the reader with a better intuition about symbolic widgets we present as an example a partial implementation of the symbolic combo widget. Figure 5 shows the source code. Symbolic combo extends the symbolic widget (line 1) and defines a concrete SWT class it represents (line 2).

```
1. public class SymbCombo extends SymbWidget {
2.   String SWT_CLASS_NAME = "org.eclipse.swt.widgets.Combo";
3.   private List<SymbSelectionListener> mSelectionListeners;
4.   private SymbString mText;
5.   . . .
6.   public SymbCombo(SymbComposite parent, SymbInteger style) {
7.     super(parent, style, "SymbCombo");
8.     . . .
9.     mText = new SymbString(20, this, “text”);
10.  }
11.  public String getSWTClassName() {return SWT_CLASS_NAME;}
12.  public void fireSelectionEvent() {
13.    SymbSelectionEvent event = new SymbSelectionEvent(this);
14.    for (SymbSelectionListener l: mSelectionListeners) {
15.      l.widgetSelected(event);
16.    }
17.  }
18.  public void addSelectionListener(SymbSelectionListener l) {
19.    TestGenerator.addELGVertex(this, EventType.SELECTION);
20.    mSelectionListeners.add(l);
21.  }
22.  public StringInterface getText() {
23.    Path.addInputVariable(text);
24.    return text;
25.  }
```

Fig. 5. Symbolic combo snippet.

The widget has a list of symbolic listeners (line 3) and a set of symbolic members representing its properties (line 4). In the constructor (lines 6-10) symbolic variables are assigned to the combo's properties (line 9). The symbolic variable receives the combo and the property it represents as parameters to associates itself with that property. The combo exposes the SWT class it represents (line 11) and defines a method for firing a selection event (lines 12-17). Client code can register event listeners (lines 18-21). Upon detection of an event listener a vertex is added to the ELG (line 19). Properties of the symbolic combo are exposed via getter/setter (setter not shown) pairs (lines 22-25). Each symbolic variable representing a widget property is added to the path (multiple additions has no effect) as an input variable (line 23), informing the constraint solver to generate an input value for this variable during the concretization phase.

5 Implementation

This section presents the components of Barad. We discuss the symbolic and concrete agents and provide an overview of the GUI testing mechanism.

5.1 Symbolic primitives, strings, and constraint solving

Barad supports symbolic operations on all primitive types (*integer*, *float*, *Boolean,* and *character*). Supported symbolic operations on integers and floats are: *and*, *or*, *addition*, *difference*, *multiplication*, *division*, *less than*, *greater than*, *greater than or equal*, and *less than or equal*. (*Booleans* are represented as integers). For solving numeric constraints Barad has a custom solver implemented via the Choco [2] library.

Supported operations on symbolic strings are: *substring*, *concat*, *charAt*, and *trim*. For symbolic string representation and constraint solving we use the work presented in [19], where finite state automata are employed to model the set of possible values for a string variable.

5.2 Barad agents

Barad consists of two collaborating agents operating on a symbolic and a concrete version of the application, respectively. They perform separate steps in the GUI testing process and can operate as stand-alone testing tools. The *Symbolic Agent* performs our algorithm for symbolic analysis and generates a test suite. The *Concrete Agent* generates and executes tests on the concrete version of the application as well as provides reports for code coverage and detected errors. While these agents operate in a collaborative fashion, test cases are generated by the *Symbolic Agent* and executed by the *Concrete Agent*. The agents run in the same Java Virtual Machine (JVM) and communicate asynchronously via publish-subscribe paradigm.

5.2.1. Symbolic agent. The Symbolic Agent instruments the GUI bytecode, performs symbolic execution of the instrumented version, and generates test cases as event sequences and data inputs. It is a Java agent that registers in the JVM for class loading events. It intercepts the loading of the main class of the AUT, instruments it, and executes it symbolically in a separate thread. Subsequently loaded classes are also instrumented at loading time.

5.2.2. Concrete agent. The Concrete Agent generates tests adopting a traditional test generation approach and executes tests on the application. In contrast with conventional GUI testing frameworks, which restart the GUI after executing a test case, the agent performs reinitialization. The agent is a JVM Tool Interface and can detect defects via uncaught exceptions thrown by the GUI at runtime.

6 Evaluation

This section presents two case studies and evaluates the applicability of our GUI testing approach. The first case study is a notepad application which does not exploit data dependent behaviors. The second case study is a workout generator program the behavior of which depends on data inputs. We compare our approach to traditional GUI testing strategies.

6.1 JNotepad

JNotepad is a Java implementation of the popular Notepad text editor. JNotepad provides basic functionalities such as creating, editing, and saving text files; cut, copy, paste, undo, redo operations etc. We analyze version 2.0 of the application. Table 3 presents a summary of JNotepad and Figure 6 shows a screenshot of the GUI.

Table 3. JNotepad application.

Windows	Widgets	LOC	Classes	Methods	Branches
8	30	849	9	51	90

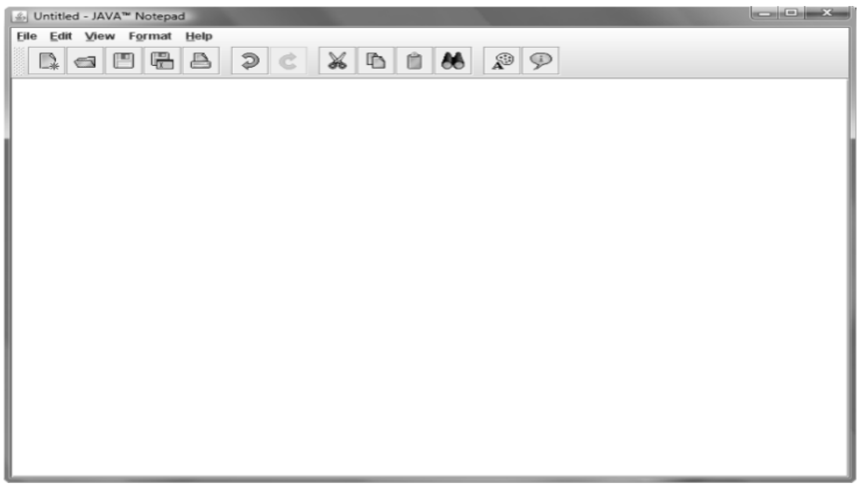


Fig. 6. Screenshot from JNotepad.

For testing JNotepad we configured Barad to ignore all widgets in the *Open*, *Save*, and *SaveAs* dialogs except the text field for specifying a file name and the *OK* and *Cancel* buttons. The file chooser class, used for implementing these dialogs, is provided by the GUI library, testing of which we want to avoid.

First, we tested JNotepad adopting our approach with enabled symbolic and event listener analysis. To limit the number of generated test cases, we configured the maximal length of event sequences before appending data populating events to three. Obtained results are presented in Table 4.

Table 4. Test results with enabled symbolic analysis.

Tests	Branch coverage, %	Statement coverage, %	Time, sec
24 058	92	97	1 495

The first column presents the total number of executed tests. The second and third columns present the branch and statement coverage, respectively. The fourth column presents the test generation and execution time (including symbolic analysis). Code coverage was reported by Barad and branch coverage was obtained by manual inspection of the code coverage report.

We next disabled the symbolic and event listener analysis simulating a traditional GUI testing approach. Values for the text boxes were selected from the set {-1, 0, 1, *Test*, *ThisIsAVeryLongStringValue*, and the empty string}. Table 5 shows the results.

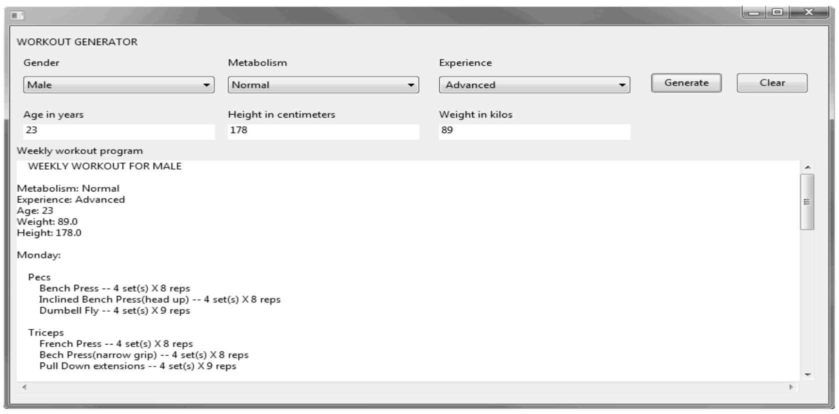
Table 5. Test results with disabled symbolic analysis.

Tests	Branch coverage, %	Statement coverage, %	Time, sec
51 694	84	91	29,46

Experimental results show that our approach generated approximately half the number of test as opposed to the traditional technique. The reason for the moderate decrease in the number of test cases generated by Barad is twofold: (1) JNotepad has few data widgets (one textbox in the main, find, and save/open windows, respectively) and does not have much data dependent behavior; (2) JNotepad contains primarily buttons, which accept a single event for which corresponding event listeners exist. Hence, for most of the events accepted by the GUI corresponding listeners exist. Despite the structure of JNotepad, which is not ideal for our technique, we still achieve significant reduction in the number of tests.

6.2 Workout Generator

The Workout Generator is a program the first author developed in his previous experience. The GUI takes as input user's biometric characteristics and generates a weekly workout program. Table 6 summarizes the characteristics of the Workout Generator and Figure 7 shows a screenshot of the GUI.



Gender: Male
Metabolism: Normal
Experience: Advanced
Age in years: 23
Height in centimeters: 178
Weight in kilos: 89

Weekly workout program

```
  WEEKLY WORKOUT FOR MALE

Metabolism: Normal
Experience: Advanced
Age: 23
Weight: 89.0
Height: 178.0

Monday:

  Pecs
    Bench Press -- 4 set(s) X 8 reps
    Inclined Bench Press(head up) -- 4 set(s) X 8 reps
    Dumbell Fly -- 4 set(s) X 9 reps

  Triceps
    French Press -- 4 set(s) X 8 reps
    Bech Press(narrow grip) -- 4 set(s) X 8 reps
    Pull Down extensions -- 4 set(s) X 9 reps
```

Fig. 7. Screenshot of the Workout Generator.

Table 6. Workout Generator application.

Windows	Widgets	LOC	Classes	Methods	Branches
1	9	651	3	15	121

The combo boxes could take one of the following values: for Gender - *Male*, *Female*; for Metabolism - *Slow*, *Normal*, and *Fast*; and for Experience - *Beginner*, *Intermediate*, and *Advanced*.

First, we tested the Workout Generator adopting our approach with enabled symbolic and listener analysis. We configured an upper bound of three for the length of event sequences. The results are presented in Table 7.

Table 7. Test results with enabled symbolic analysis.

Tests	Branch coverage, %	Statement coverage, %	Time, sec
48	100	100	4.3

We next disabled the symbolic and event listener analysis simulating a traditional GUI testing approach. The values for data widgets were chosen as follows: for text-boxes a value from the set {-1, 0, 1, *Test*, *ThisIsAVeryLongStringValue*, and the empty string}; for combo-boxes, a value from the set of possible values. We set the maximal length of generated event sequences to three. The results are presented in Table 8.

Table 8. Test results with disabled symbolic analysis.

Tests	Branch coverage, %	Statement coverage, %	Time, sec
5 984	76	97	285

Experimental results show that for the Workout Generator our approach generates significantly fewer test compared to the traditional technique. The reason for that is twofold: (1) Workout Generator has a fair amount of data widgets and exploits data dependent behaviors; (2) Workout Generator has fewer listeners. The structure of the Workout Generator is opportune for our technique and we achieve in order of two magnitudes decrease in the number of test.

7 Discussion

The experimental results show that our approach generates fewer tests and achieves higher branch and statement coverage compared to traditional GUI testing techniques. Further, our approach addressed data-flows in GUI applications by generating inputs for data widgets, which force the execution of different program paths. Our technique is especially effective for testing data intensive GUI applications, with data dependent behavior.

Since we perform symbolic analysis, our technique inherits the limitations of symbolic execution with regard to native calls. While our implementation does not handle native calls we can adopt the approach for approximation symbolic execution presented in [16]. Another issue that arises during symbolic execution is handling of loops. We take a standard approach and perform loop unwinding up to a given bound. Such an approximation inevitably introduces errors. Further, symbolic execution requires solving of path constraints, which in the general case, are undecidable.

The current implementation of Barad supports a subset of the SWT GUI library which prevents us to apply our approach to the written with Swing TerpOffice, an application suite used by Memon et al. in his extensive work in GUI testing.

We currently detect bugs as runtime exceptions. However, specification based oracles that check richer properties would enable more thorough testing of GUIs. We do not report detected bugs since we adopt the same fault detection strategy as the conventional GUI testing performed by Memon et al. Our focus is on reducing test suite size and improving statement and branch coverage.

8 Related work

To the best of our knowledge, in his Ph.D. dissertation [9] Memon presents the first framework for GUI testing that generates, runs, and assesses GUI tests. The framework focuses on the event-flow of GUI applications. For emulating user input a specification based approach is adopted—using values from a prefilled database. The components of the framework and its extensions are presented in several papers [9], [11], [13], [14], [22]. This framework considers all events accepted by the GUI while we focus on events with event listeners. The framework does not provide a mechanism for obtaining inputs for data widgets. By providing such a mechanism our work is complementary in this respect.

Memon, Banarjee and Nagarajan present a framework for regression testing of nightly/daily builds of GUI applications [12]. This tool addresses rapidly evolving GUI applications executing a small enough test suite that the test process could be accomplished in less than a day/night. This framework is based on the one presented in Memon's PhD dissertation [9] and uses the same test generation algorithm and specification based approach to simulate user inputs. We employ a different test generation algorithm and present a technique for obtaining data inputs.

Another approach is representing the GUI as a Variable Finite State Machine from which after a transformation to an FSM, tests are obtained [18]. This black-box testing technique does not consider user input while focusing on the event-flow. Our approach is white-box with dynamic analysis focusing on event listeners and generates data inputs.

A technique for testing a GUI is transforming the GUI into a FSM and using different techniques to reduce the states of that FSM to avoid state space explosion [21]. In approach the focus is on collaborating selections and user sequences over different objects in the GUI. This is a white-box event centric approach that abstracts away user inputs. We adopt an event listener centric technique and generate data values.

Verification of GUI specifications has been performed via model checking [3]. The authors introduce domain specific abstractions to reduce the state space to be explored. The GUI and its behavior are represented as a Computation Tree Logic in the input language of the SMV model checker via a manual process. In contrast, our approach is fully automatic and aims at test generation rather than at model checking. We see this work as complementary to our approach.

A technique for updating test scripts for evolving GUI applications has been proposed [6]. This enables reuse of existing scripts via detecting script errors due to changes in the GUI. Our work focuses on test generation and is complementary.

A system that automatically extracts a program interface, generates a test driver and a random test suite after completion of which symbolic execution is used to guide the generation of additional tests has been presented [15]. Similarly, we employ symbolic execution to generate tests which maximize coverage by exploring different program paths. We introduce the abstraction of symbolic widgets which allows scaling symbolic execution for GUIs.

Symbolic execution and concrete execution have been combined for test generation [16]. This approach uses approximate symbolic execution for testing code with dynamic data structures. In contrast, we generate inputs in the form of string and numeric data and do not perform concrete execution. We take advantage of the systematic approach for path exploration and scale symbolic execution for GUIs.

Symbolic execution has been used for test data generation [23]. The program is represented as a deterministic FSM and using symbolic execution generates test data. This work deals exclusively with numeric constraints. Barad performs symbolic execution over GUI components (widgets) and strings (in addition to primitives).

9 Conclusion

We presented Barad, a novel GUI testing framework that addresses event-flow as well as data-flow for white-box testing of GUI applications. Barad is fully automatic, performing instrumentation, symbolic execution, test generation, and test execution.

We introduce the abstraction of symbolic widgets. This abstraction enables symbolic analysis to reason about the control flow in GUI applications without analyzing the GUI library implementation. We generate test cases as sequences of events with registered listeners, pruning significant regions of the event input space. We execute symbolically the sequence of listeners registered for the events in a test case enabling a systematic approach to obtain inputs for data widgets.

We evaluate our framework on non trivial GUI subjects. Compared to traditional GUI testing techniques Barad achieves higher statement and branch coverage while generating significantly fewer tests.

Acknowledgements. This material is based upon work partially supported by the NSF under Grant Nos. IIS-0438967, CCF-0702680, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

10 References

1. ASM: Java bytecode manipulation and analysis framework, http://asm.objectweb.org/
2. Choco: Java library for constraint solving, http://sourceforge.net/projects/choco/
3. Dweyer, M., Carr, V., Hines, L.: Model Checking Graphical User Interfaces Using Abstractions. In *ESEC*, 1997.
4. Emma: Java code coverage tool, http://emma.sourceforge.net/
5. Ganov, S., Killmar, C., Khurshid, S., Perry, D., E.: Test Generation for Graphical User Interfaces Based on Symbolic Execution. In *AST*, 2008.
6. Grechanik, M., Xie, Q., Fu, C.: Maintaining and Evolving GUI-Directed Test Scripts. In *ICSE*, 2009.
7. King, J.: Symbolic Execution and Program Testing. In *Communications of the ACM*, 1976.
8. Lori, C.: A System to Generate Test Data and Symbolically Execute Programs. In *IEEE Transactions on Software Engineering*, 1976.
9. Memon A.: A Comprehensive Framework For Testing Graphical User Interfaces. In *Ph.D. Thesis, University of Pittsburgh*, 2001.
10. Memon, A.: Using Tasks to Automate Regression Testing of GUIs. In *AIA*, 2004.
11. Memon, A., Banarjee, I., Nagarajan, A.: GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *WRCE*, 2003.
12. Memon, A., Banarjee, I., Nagarajan, A.: DART: A Framework for Regression Testing Nightly/Daily Builds of GUI Applications. In *ICSM*, 2003.
13. Memon, A., Banarjee, I., Nagarajan, A.: What Test Oracle Should I use for Effective GUI Testing? In *ASE*, 2003.

14. Memon, A., McMaster, S.: Call Stack Coverage for GUI Test-Suite Reduction. In *ISSRE*, 2006.
15. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In *PLDI*, 2005.
16. Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE'05* 2005.
17. Ramamoorthy, V., Siu-Bun, H., Chen, W.: On the Automated Generation of Program Test Data. In *IEEE TSE*, 1976.
18. Shehady, R., K., Siewiorek, D.: A Method to Automate User Interface Testing Using Variable Finite State Machines. In *FTCS*, 1997.
19. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting Symbolic Execution with String Analysis. In *TAICPART-MUTATION*, 2007.
20. SWT: The Standard Widget Toolkit, http://www.eclipse.org/SWT
21. White, L., Almezen, H.: Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *ISSRE*, 2000.
22. Xie, Q., Atif, M.: Using a Pilot Study to Derive a GUI Model for Automated Testing. In *TOSEM*, 2008
23. Zhang, J., Xu, C., Wang, X.: Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques. In *SEFM*, 2004