Barad – A GUI Testing Framework Based on Symbolic Execution

Svetoslav Ganov[1], Chip Kilmar[2], Sarfraz Khurshid[2], Dewayne Perry[1]
[1] Laboratory of Experimental Software Engineering, University of Texas at Austin
[2] Software Testing and Verification Group, University of Texas at Austin
{svetoslavganov, ckillmar}@mail.utexas.edu, {khurshid, perry}@ece.utexas.edu

Abstract

While Graphical User Interfaces (GUIs) have become ubiquitous, testing them remains largely ad-hoc. Since the state of a GUI is modified by events on the GUI widgets, a useful approach is to consider test input for a GUI as an event sequence. Due to the combinatorial nature of these sequences, testing a GUI thoroughly is problematic and time-consuming. Moreover, the possible values for certain GUI widgets, such as a textbox, are also combinatorial compounding the problem.

This paper presents Barad, a novel GUI testing framework based on symbolic execution. Barad addresses uniformly event-flow as well as data-flow in GUI applications: generating tests in the form of event sequences and data inputs. We generate test cases as chains of event listener method invocations and map these chains to event sequences that force the execution of those invocations. Since listeners for some events in the GUI are not present, this approach prunes significant regions of the event input space. We introduce symbolic widgets as a higher level of abstraction, which enables symbolic execution of GUI applications. We obtain data inputs through executing symbolically the generated test cases (chains of event listener method invocations). Barad generates significantly fewer tests compared to traditional GUI testing techniques, while improving branch and statement coverage.

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. While they have become ubiquitous, testing them remains largely ad-hoc. In contrast to console applications where there is only one point of interaction (the command line), GUIs provide multiple points each of which might 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 GUI input. Exhaustive testing requires trying all 120 possible combinations since triggering of one event before another may cause execution of different code segments.

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 1026 possible input strings (assuming we limit each character to be only alphabetical in lower-case).

Automation of GUI testing has traditionally focused on minimizing event sequences [9] [12] [16] [19]. Data widgets have either been abstracted away by not considering GUI behaviors dependent on data values, populated by values generated at random, or selected from a manually constructed set consisting of a small number 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.

This paper presents Barad, a novel GUI testing framework based on symbolic execution [7] [8] [15] for checking GUI applications written in Java with the *Standard Widget Toolkit* (SWT) [18] library. Barad generates event sequences and input values for data widgets providing a systematic approach that uniformly addresses event-flow as well as data-flow for white-box testing of GUI applications. 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 (AWT, Swing, SwingWT, BambooKit, etc).

To scale symbolic execution for GUI applications, we introduce the abstraction of *symbolic widgets* – entities that enable symbolic manipulation of standard GUI widgets. GUI widget implementations have three concerns: (1) functionality; (2) visualization; (3) and 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.

Barad handles event-flows in GUI applications by using the symbolic widgets to detect *event listeners* – instances that register for and respond to events in the GUI. We generate test cases as chains of event listener method invocations and map these chains to event sequences that force the execution of these invocations. Our approach prunes significant regions of the event input space because we don't have to consider events where there are no event listeners.

Barad handles data-flows in GUI applications by symbolically executing the generated test cases (chains of event listener method invocations). During symbolic execution, feasible program paths are systematically explored and (for decidable constraints) infeasible paths are detected. For each feasible path concrete inputs are generated. These inputs are values for the fields of GUI widgets and define a GUI state that forces the execution of a particular program path. Reaching this state requires an event sequence to be executed in the GUI. This sequence includes events that populate values of data widgets and events that perform actions on *action widgets* – widgets that accept user input in the form of actions (buttons, etc).

We make the following contributions:

- **Symbolic execution 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. We define a test case as a chain of even listener method invocations, pruning significant regions in the event input space.
- **Data input generation.** By symbolically executing test cases we obtain inputs for data widgets, pruning significant regions in the data input space.
- **A novel 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 techniques for GUI testing. Barad generates significantly fewer tests and achieves higher statement and branch coverage.

2. Example

In this section we provide an overview of our GUI testing approach and demonstrate how it uniformly handles event-flows and data-flows in GUI applications. Further, we compare our approach 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*.

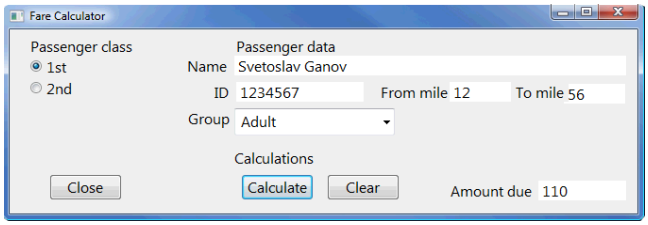

Figure 1. Screenshot of the Fare calculator

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 selection events in buttons *Close*, *Calculate*, and *Clear*, respectively. The main computation is performed by the event listener registered for responding to the *Calculate* button. The calculation logic has twenty two 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 (i.e. selecting a radio button).

2.2. Test results

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 GUI widget and one value per data widget is slightly less than 4,000,000 (11!). Further, 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 (1010) increase in the test suite size. Hence, due to the two dimensional combinatorial explosion in GUI inputs, exhaustive

testing of even as simple a GUI as the Fare Calculator is unrealistic. Clearly, a systematic approach that prunes regions of the event and data input space is required for practical testing.

2.3. Input space

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, %	Line, %	Time, sec
246	100%	100%	10.34

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 [5] 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 selecting each of the buttons), which correspond to fifteen possible chains of event listener method invocations up to length three, without repetition. Each chain is symbolically executed and for some chains sets of input values were determined. Each chain was mapped to an event sequence which forces method executions, defining a candidate test case. Candidate test cases, whose corresponding chains generated sets of input values, were prefixed with events to populate each set of values, producing a new test case. The full branch coverage is due to data values obtained by systematic exploration of all feasible paths during symbolic execution.

Conventional GUI testing techniques [9] [12] [16] [19] generate exhaustively 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. Further, we limit the upper bound for the length of event sequences to be equal to the upper bound for the length of generated chains of event listener method invocations. 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*, and 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.

2.4. Comparison

The test results show that our approach generates more than two orders of magnitude fewer tests compared to a traditional approach, while achieving significantly higher branch coverage. The longest event sequence generated by our approach has length eight and consists of the following events: (1) selection of 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; In contrast, generating test cases with length up to eight events without repetition via the traditional approach using the very limited input specifications results 7.6 x 10^{13} test cases, the execution of which is unrealistic.

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 3, which swaps the values of integer variables x and y, when x is greater than y. Figure X 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.

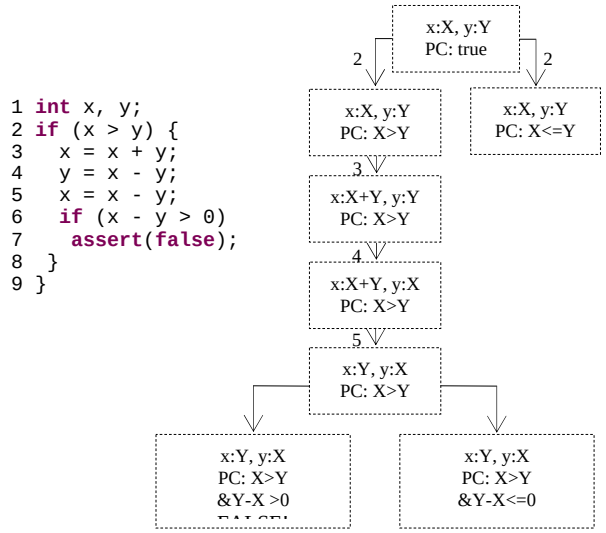

Figure 3. Code that swaps two integers and the its symbolic execution tree

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 done.

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 [17]. 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 each 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.

A GUI state is a triple (W, ρ, ω) that consists of a set of widgets, a mapping $\rho : W \rightarrow 2^P$ from widgets to properties, and a mapping $\omega : P \rightarrow V$ from properties to values.

Each GUI widget accepts as input a set of user events E triggered by user actions. Examples of events are *clicks*, *mouse moves*, etc. Formally, events accepted by a GUI widget are defined as:

$$\forall w \in W \mid \exists E_w \supseteq E : accept(w, E_w)$$

Each GUI widget has zero or more event listeners L registered for events performed on the widget. Event listeners contain computational logic and are notified (their methods invoked) when the corresponding event occurs. Examples of listeners are *selection listener*, *modification listener*, etc. Formally, event listeners are defined as:

$$\forall w \in W \mid \exists L_w \supseteq L \wedge \forall l \in L_w \mid \exists E_l \supseteq E_w \wedge \forall e \in E_l \mid reg(l, e)$$

Since the user interacts with the GUI through events, a GUI test case (from the set T of GUI test cases) is an event sequence. Formally, a GUI test case is defined:

$$\forall t \in T : t = < e_1, e_2, ..., e_p >$$

4. Barad

This section presents Barad, our GUI testing framework. We present the adopted abstractions and processes enabling systematic testing of GUI applications using symbolic execution.

4.1. GUI testing process in Barad

The process of GUI testing performed by Barad is shown on Figure 4. 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

[2]. 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. During this process event listeners are detected, chains of event listener method invocations are generated, and then symbolically executed – i.e., all paths are systematically explored and their feasibility evaluated by constraint solving.

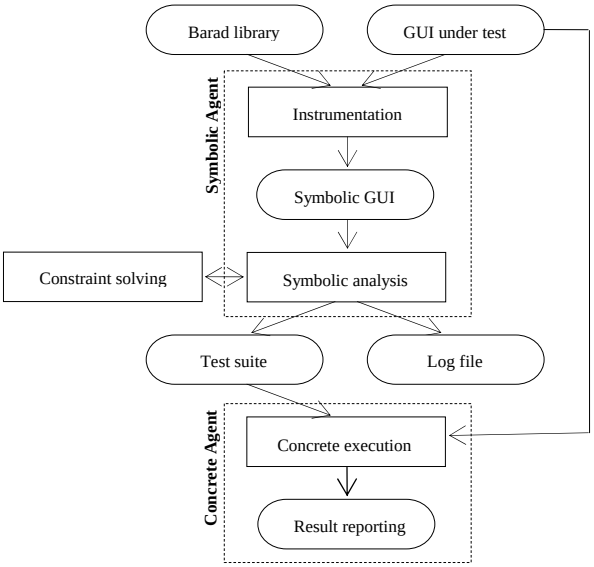

Figure 4. GUI testing process in Barad

As a result of this process a log file and a test suite are generated. The log file contains information about the constraints for each explored path, its feasibility, and concrete values for the GUI's data widgets (if the path is feasible) that would enforce the execution of this path.

The test suite consists of chains of event listener method invocations mapped to event sequences and concrete inputs. Such a test suite maximizes code coverage while minimizing the number of tests needed to systematically check the GUI. Finally, the test suite is executed on the concrete version of the application and a coverage report is generated.

4.2. Event listener count versus event count

While the number of events accepted by a GUI widget is constant, the number of event listeners, registered for events in the widget, can vary. As an illustration consider the Fare Calculator from Section 2.

The GUI consists of textboxes (*org.eclipse.swt.widgets.Text*), buttons (*org.eclipse.swt.widgets.Button*), and a combo (*org.eclipse.swt.widgets.Combo*). Each of these widget classes extends *org.eclipse.swt.widgets.Control*, which receives the following set of events: *ControlEvent*, *FocusEvent*, *HelpEvent*, *KeyEvent*, *MouseEvent*, *PaintEvent*, and *TraverseEvent*. In addition, there are some widget specific events: buttons accept *SelectionEvent*; textboxes accept *SelectionEvent*, *ModifyEvent*, and *VerifyEvent*; combos accept *SelectionEvent* and *ModifyEvent*. Table 2 presents the widget specific events for the Fare Calculator.

Widge	Possible	Widgets	Total
Button	1	5	5
Text	2	6	12
Combo	2	1	2

Table 2. Possible events for the Fare Calculator

The first column presents the widget type. Columns, two, three, and four present the number of possible events per widget type, the total number of widgets, and the total number of events, respectively.

We consider only the widget specific events since they are the most widely used in practice (i.e. have corresponding event listeners) and are the focus traditional GUI testing techniques. The data in the table shows that for the nineteen possible widget specific events, the Fare calculator has only three event listeners. Hence, the possible combinations of event listener method invocations are significantly fewer than the possible event sequences. Hence, focusing our analysis on event listener we potentially prune large regions in the event input space.

4.3. Symbolic widgets

To enable symbolic execution of GUI applications, 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 and symbolic data, respectively. For example, the *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 in the 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) We use the symbolic widgets to detect event listeners at run time. Detecting of event listener is required by our test generation algorithm. (4) Symbolic widgets enable us to detect the mapping from events to event listeners, which is used in the transition from chains of event listener method invocations to event sequences. (5) 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 chains of event listener method invocations. (6) Similarly to their concrete counterparts, symbolic widgets are referenced by the events passed as parameters to the event listeners. This provides a mechanism of accessing properties of symbolic widgets through the 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. (3) We avoid symbolic analysis of native calls. The inability to handle native calls is a well known limitation of symbolic execution.

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.

4.4. Test generation algorithm

Taking advantage of the symbolic widgets we developed our test generation algorithm shown in Figure 5.

We represent the GUI event listener model as an *Event Listener Graph* (ELG)—a directed graph with nodes representing event listeners and edges. The existence of an edge from event listener *e1* to event listener *e2* means an execution of event listener *e2* can be performed immediately after the execution of event listener *e1*. For example, if event listener *e1* opens a new *form* (GUI window) every event listener in that form strictly succeeds *e1*. Every time a new event listener is identified a new node is added to the graph. Since event listeners are detected () by the symbolic widgets in which they are registered (i.e. at runtime) and these listeners can open other forms (containing other widgets with registered event listeners), all event listeners in the GUI should be executed at least once (line 2) 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 chains of event listener method invocations up to a given bound without repetition. Each of these chains is then mapped to a corresponding event sequence resulting in a candidate test case *t*. As a result a set $T = \{t_1, t_2, \ldots, t_n\}$ of candidate tests is obtained (line 4).

We obtain data inputs by symbolically executing every chain of event listener method invocations obtained from the previous step (line 6-11). Doing so, we capture data dependencies between the event listeners. For each chain we potentially identify sets of input values for the data widgets in the GUI (line 8). 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 corresponding to the event listener chain (line 10).

```
1  //build ELG
2  SymbolicModel.executeListeners();
3  //generate event listener chains
4  chains = TestGenerator.generateChains();
5  //execute symbolically each chain
6  for (Chain c: chains){
7    //obtain data inputs for each chain
8    in = SymbolicModel.excecuteChain(c);
9    //generate test - append inputs to chains
10   test.addAll(TestGenerator.generateTests(c,in);
11 }
```

Figure 5. Test generation algorithm

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 listener in the GUI (line 2). As a result form this step all three event listeners (registered for selection the three buttons) $l1$, $l2$, and $l3$ are identified and used for construction of fifteen event listener method invocation chains (line 4). These chains are symbolically executed (line 8). Without loss of generality, consider the chain ($l1$, $l2$, $l3$) the execution of which generated twenty two sets S of five inputs values $v1 - v5$ each:

$$(l1, l2, l3) \rightarrow \{S1\{v1,..,v5\}, S2\{v1,..,v5\},....,S22\{v1,..,v5\}\}$$

Each input set transitions the GUI to such a state that executing the chain ($l1$, $l2$, $l3$) 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 events that execute the chain of event listeners (line 10). The generated test cases, where $e(x)$ is the event required for populating the value x or triggering the event listener x, look as follows:

$e(v1^{S1}), e(v2^{S1}), \ldots, e(v5^{S1}), e(l1), e(l2), e(l3);$
…
$e(v1^{S22}), e(v2^{S22}), \ldots, e(v5^{S22}), e(l1), e(l2), e(l3);$

5. Implementation

This section presents the main components of Barad. We discuss the symbolic and concrete agents, which compose the system.

5.1. Symbolic primitives and strings

Barad support symbolic operations on all primitive types (*integer*, *float*, *boolean*, and *character* - represented as a string with length one). Supported symbolic operation 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 symbolic string representations we use the work presented in [23], where they employ finite state automata to model the set of possible values for a string variable.

5.1. Barad Agents

Barad consists of two collaborating agents operating on a symbolic and a concrete version of the application under test (AUT), 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 AUT 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 the publish-subscribe paradigm.

5.1.1. Symbolic agent. The *Symbolic Agent* instruments the bytecode of the AUT, 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 of the AUT are also instrumented at loading time.

The main components of the *Symbolic Agent* are: (1) *Custom class loader*—enables parallel execution of a symbolic and a concrete version of the AUT in the same JVM; (2) *Instrumenter*—performs the Java bytecode manipulation; (3) *Symbolic Analyzer*—performs symbolic and event listener analysis; and (4) *Test generator*—generates tests from the data obtained during the symbolic and event listener analysis;

5.1.2. Concrete agent. The *Concrete Agent* generates tests adopting a traditional test generation approach and executes tests on the AUT. In contrast with conventional GUI testing frameworks, which restart the GUI after executing a test case, the agent reinitializes the AUT. The agent uses a client-server architectural. It is a JVM Tool Interface (JVMTI) and can detect defects via uncaught exceptions thrown by the AUT at runtime.

The main components are: (1) *Test generator*—generates test; (2) *Test executor*—executes test on AUT; and (3) *Barad Studio*—presents visualization aids and controls the testing process;

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[1] 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

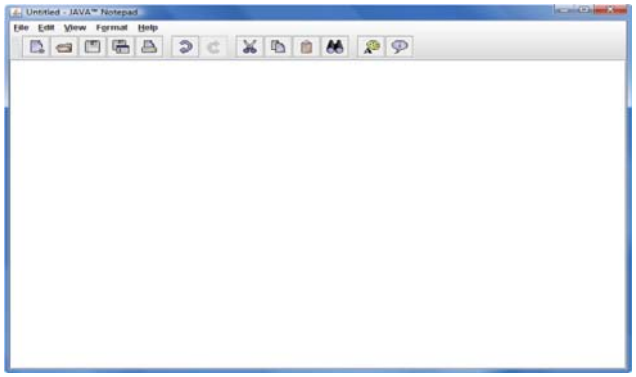

Figure 6. Screenshot from JNotepad

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 of 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 listener method invocation chains to three. Obtained results are presented in Table 4.

Table 4. Test results with enabled symbolic analysis

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

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. The 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, %	Line%	Time, sec
51,694	84%	91%	29,465 sec

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. This makes the number of events close to the number of listeners.

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. 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*.

Table 6. Workout Generator application

Windows	Widgets	LOC	Classes	Methods	Branches
1	30	649	3	15	121

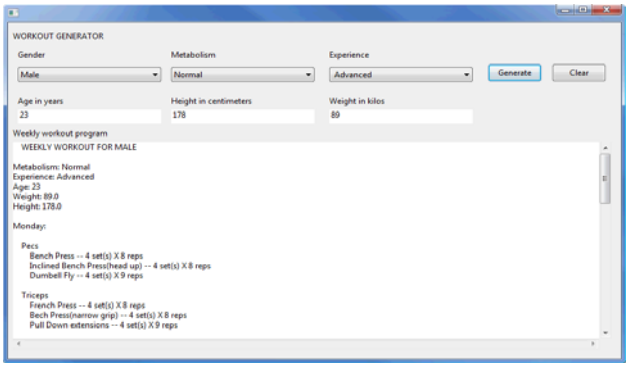

Figure 7. Screenshot from Workout Generator

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 the chains of event listener method invocations. The results are presented in Table 7.

Table 7. Test results with enabled symbolic analysis

Tests	Branch, %	Line, %	Time, sec
48	100%	100%	3 sec

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 form the set of possible values. We set the maximal length of generated event sequences to three. The results are presented in Table8.

Table 8. Test results with disabled symbolic analysis

Tests	Branch, %	Line, %	Time, sec
37,060	76%	97%	1762 sec

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 three 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 regards to native calls. By abstracting away the visualization layer of GUI widgets, we avoid the native calls to the operating system. 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 in the generated concrete values. 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. Our objective was to verify the applicability of our approach.

8. Related work

To the best of our knowledge previous work in GUI testing has not considered symbolic execution. In a workshop poster last year, we introduced the idea of symbolic execution for GUI testing. Our recent publication [22] discusses unit testing of GUI event listeners in isolation. This paper presents Barad, a comprehensive framework that executes symbolically GUI application and considers event listeners to be first class entities during the test generation phase.

In his Ph.D. Dissertation [9] Memon presents a framework for GUI testing that generates, runs, and assesses GUI tests. This was the first framework capable of performing the whole process of test generation, execution, and result assessment for GUIs. His 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 main components of the framework are presented in [9], [10], [11], [13]. The most recent research based on this framework is presented in [20] by Memon and Xie. This framework generates tests as event sequences up to a given bound while we define tests as chains of event listener method invocations mapped to such event sequences. The framework does not provide a mechanism for obtaining input values for data widgets, and 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 [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 inputs.

Another approach is that in which a GUI is represented as a Variable Finite State Machine from which after a transformation to an FSM, tests are obtained [16]. This approach does not consider user input while focusing on the event-flow of GUIs. Again this is a black-box testing approach focusing on events as first class entities during the test generation phase and user inputs are not considered. Our approach is a white-box approach with dynamic analysis focusing on

event listeners and providing technique for input generation.

A technique that transforms GUIs into a FSM and uses different techniques to reduce the states of that FSM to avoid state space explosion is proposed in [19]. In this work the focus is on collaborating selections and user sequences over different objects in the GUI. This approach is similar to the one presented in [16]. It is a white-box event centric approach that abstracts away user inputs. We adopt an event listener centric approach and generate data values.

Symbolic execution for test data generation is used in [21]. 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). Also the input variables for GUI event listeners (symbolically executed during our analysis) have multiple entry points as opposed to this approach where input variables have a single entry point—the method parameters.

The Java String Analyzer [4] performs static analysis of Java programs and generates a context-free grammar for each string expression represented as a multilevel automaton. Barad uses similar approach to dynamically build a finite state automaton for each string variable that accepts only values that satisfy the path conditions.

9. Conclusion

We presented Barad, a novel GUI testing framework based on symbolic execution that addresses event-flows as well as data-flows 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 flows in GUI applications without analyzing the GUI library implementation. We generate test cases as chains of event listener method invocations, pruning significant regions of the event input space. We execute symbolically the generated tests 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.

10. References

[1] A. Møller. Brics automaton library. http://www.brics.dk/automaton.

[2] ASM, Retrieved on November 1, 2007, http://asm.objectweb.org/

[3] Choco, Retrieved on January 29, 2008 from Sourceforge:

[4] Christensen, A., S., Møller, A., and Schwartzbach, M., I. Precise Analysis of String Expressions. SAS 2003, 1-18, 2003.

[5] Emma, Retrieved on January 29, 2008, http://emma.sourceforge.net/

[6] GuillaumeBrat, Klaus Havelund, SeungJoon Park and Willem Visser '*Java PathFinder, Second Generation of a Java Model Checker*', Research Institute for Advanced Computer Science, 2000.

[7] King, J., Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.

[8] Lori, C., A system to generate test data and symbolically execute programs, IEEE Transactions on Software Engineering, 2(3):215-222, September 1976.

[9] Memon, A. A comprehensive Framework For Testing Graphical User Interfaces. Ph.D. Thesis, University of Pittsburgh, Pittsburgh, 2001.

[10] Memon, A. Using Tasks to Automate Regression Testing of GUIs. In International Conference on Artificial intelligence and Applications (AIA 2004), Innsbruck, Austria, Feb. 16-18, 2004.

[11] Memon, A., Banarjee, I., and Nagarajan, A. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In Reverse Engineering, 2003, WRCE 2003. Proceedings. 10th Working Conference on, November 13-16, 2003, 260-269.

[12] Memon, A., Banarjee, I., and Nagarajan, A. "DART: A Framework for Regression Testing Nightly/Daily Builds of GUI Applications". In International Conference on Software Maintenance 2003 (ICSM'03), Amsterdam, The Netherlands, Sep. 22-26, 2003, pages 410-419.

[13] Memon, A., Banarjee, I., and Nagarajan, A. What Test Oracle Should I use for Effective GUI Testing?. In IEEE International Conference on Automated Software Engineering (ASE'03), Montreal, Quebec, Canada, Oct. 6-10 2003, pages 164-173.

[14] Memon, A., and McMaster, S. Call Stack Coverage for GUI Test-Suite Reduction. In Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006), Raleigh, NC, USA, Nov. 6-10 2006.

[15] Ramamoorthy, V., Siu-Bun, H., and Chen, W., On the automated generation of program test data, IEEE TSE, 2(4):293-300, 1976.

[16] Shehady, R., K., and Siewiorek, D., P. A Method to Automate User Interface Testing Using Variable Finite State Machines. In 27th International Symposium on Fault-Tolerant Computing, p. 80, 1997.

[17] Squish, Retrieved on January 25, 2008 from FrogLogic http://www.froglogic.com/

[18] SWT: The Standard Widget Toolkit, 2007. Retrieved October 15, 2007, http://www.eclipse.org/SWT

[19] White, L., and Almezen, H. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *11th ISSRE*, p.110, 2000.

[20] Xie, Q., and Atif M. Memon, Using a Pilot Study to Derive a GUI Model for Automated Testing *ACM Trans. on Softw. Eng. and Method.*, 2008

[21] Zhang, J., Xu, C., and Wang, X. Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques. In *Software Engineering and Formal Methods (SEFM 2004)*, p.242-250, 2004

[22] Ganov, S., Killmar, C., Khurshid, S., Perry, D., Test Generation for Graphical User Interfaces Based on Symbolic Execution, *Third International Workshop on Automation of Software Testing (AST)*, Leipzig, Germany 2008

[23] Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S., Abstracting Symbolic Execution with String Analysis Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007