Test Generation for Graphical User Interfaces Based on Symbolic Execution

Svetoslav Ganov
The University of Texas at Austin
Laboratory of Experimental Software Engineering
USA, Austin, Texas 78712
+1 832 366-4884
svetoslavganov@mail.utexas.edu

Chip Killmar
iTKO Inc
USA, Dallas, Texas 75234
+1 512 415-4256
ckillmar@mail.utexas.edu

Sarfraz Khurshid
The University of Texas at Austin
Software Testing and Verification Group
USA, Austin, Texas 78712
+1 512 471-8244
khurshid@ece.utexas.edu

Dewayne E Perry
The University of Texas at Austin
Laboratory of Experimental Software Engineering
USA, Austin, Texas 78712
+1 512 471-2050
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 defined by a sequence of events on the GUI's widgets, a test input for a GUI is such an event sequence. Due to the combinatorial nature of the sequences, testing a GUI thoroughly is problematic and time-consuming. Moreover, the wide range of possible values for certain GUI widgets, such as a textbox, compounds the problem.

This paper presents a novel test generation approach based on symbolic execution to obtain data inputs and enumerate event sequences that are likely to maximize code coverage of a GUI application. Key contributions are introducing the technique of symbolic execution in GUI testing (addressing a common weakness of traditional GUI testing frameworks) and performing symbolic execution over strings (in addition to primitives). Doing so minimizes the number of event sequences that form the resulting test suite. To determine feasibility of path conditions that arise in symbolic execution, we implement a solver for constraints over strings (in addition to primitives). We evaluate our test generation approach.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and debugging – Symbolic Execution, Testing Strategies, Testing Tools

General Terms
Verification, Reliability

Keywords
GUI testing, symbolic execution, test input generation

1. INTRODUCTION
A Graphical User Interface (GUI) is a convenient way to interact with the computer. It consists of virtual objects (widgets) that are more intuitive to use, for example buttons, edit boxes, etc. While GUIs have become ubiquitous, testing them remains largely ad-hoc. In contrast with console applications where there is only one point of interaction (the command line), GUIs provide multiple points each of which might have different states. This structure makes GUI testing especially challenging because of its large input space.

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. To illustrate, 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 because in the internal logic of the GUI, 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 are used for user input, such as textboxes, edit-boxes and combo-boxes, and can have an extremely large space of possible values. To illustrate, consider testing a GUI with one textbox that takes a ten character string as an input. Exhaustive testing requires 10^{26} possible input strings (assuming we limit each character to be from the English alphabet in lower-case).

Automation of GUI testing has traditionally focused on minimizing the event sequences. Data widgets have either been abstracted away by not considering GUI behaviors dependent on data values, or populated by values generated at random, or selected from a manually constructed set consisting of a small number of values [7] [10] [14] [15]. As a consequence, data dependent behaviors are inadequately tested. For example, 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 a tedious code inspection and does not scale. A specification-based (black-box) approach may find this "special" value, however it would require detailed specifications, which are often not feasible to write and often not provided.

This paper presents Barad, a novel GUI testing framework based on symbolic execution [5] [6] [13]. Barad generates values for data widgets and enables a systematic approach that uniformly addresses the data-flow as well as event-flow for white-box testing of a GUI application. We symbolically execute the code of GUI event handlers and generate data inputs that maximize code coverage while minimizing the number of tests needed to systematically check the GUI.

During symbolic execution all reachable paths of the program are systematically explored and (for decidable constraints) infeasible

Conference'04, Month 1–2, 2004, City, State, Country.

paths are detected. For each feasible path data inputs for the GUI widgets are generated. Because of the multiple points of interaction of GUIs these inputs may include the current state of several GUI widgets. These widgets could be not only data widgets but also *sequence widgets*—widgets used for user input in the form of event sequences (buttons, check-boxes, radio-buttons). Identifying a state of the GUI that allows us to execute a program path (selected radio-button) defines an event sequence (selecting the radio-button) that should be applied to the GUI to reach this state. This way our approach also addresses the event-flow of GUI testing.

We make the following contributions:

- **Symbolic execution**—we explore the applicability of symbolic execution for systematically testing GUI applications by generating data inputs for the GUI widgets by analyzing GUI event handlers.
- **Algorithm**—we present an algorithm that generates GUI input values and implements efficient solvers for strings and primitives.
- **Test reduction**—our algorithm minimizes the number of the generated inputs while (ideally) preserving the code coverage.
- **Implementation**—we have implemented our algorithm in a prototype Barad for testing of Java applications. Barad performs analysis of a GUI application fully automatically by instrumenting its Java bytecode.

2. EXAMPLE

This section provides the basics of how our technique is applied to GUI testing. The goal is to show an example where conventional GUI testing techniques would most probably achieve low coverage unless a prohibitively large test suite is used.

The GUI presented in Figure 1 is a program (300 lines of code) we developed to demonstrate our approach. It calculates the amount due for a train ticket. A user must provide a passenger class, name, ID, group, and begin and end points. The passenger groups are *Senior*, *Adult*, *Student*, and *Child*.

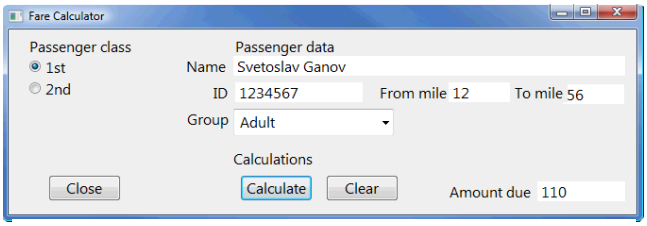

Figure 1. Example application

Each passenger class has its own coefficient that is used during the calculation. Each group has different base price depending on the distance to be traveled, which is the difference between the values in the text-boxes *From mile* and *To mile*. The application has multiple branches the execution of which depends on user input both in the form of input data and event sequence (i.e. selecting a radio button). Figure 2 shows a code fragment for the *Senior* group. The calculation method has twenty-two branches with conditions nested three levels.

```
. . .
1  int distanceRange =
2     Integer.parseInt(text3.getText().trim())-
3     Integer.parseInt(text4.getText().trim());
4  String class =  combo1.getText().trim();
5  if (group.equals("Senior")) {
6    if (distanceRange < 50) {
7      amountDue = 120 * coeficient;
8    } else if (distanceRange < 60) {
9      amountDue = 130 * coeficient;
19   } else if (distanceRange < 70) {
11      amountDue = 145 * coeficient;
12   } else if (distanceRange < 80) {
13      amountDue = 150 * coeficient;
14   } else if (distanceRange < 100){
15      amountDue = 160 * coeficient;
16   }
17 } else if (group.equals("Adult")) {
. . .
```

Figure 2. Code fragment for the *Senior* group

Using Barad a test suite is generated and run on the GUI. Results are presented in Table 1. The full branch and code coverage are not surprising since symbolic execution guarantees visiting of all reachable branches.

Table 1. Results of symbolically generated test suite

Tests	Branch Coverage	Code Coverage	Generation Time
22	100%	100%	4.34 sec

We next compare our approach to random test generation. Our methodology is as follows. For the passenger class we select randomly from—*1st*, and *2nd*. For the group a random choice is made out of the four possible values in the combo-box. The input for the text-boxes *From mile* and *To mile* is a number between zero and ninety-nine chosen randomly. For randomization, the Java random class is used. Fifty test suites, each with different seed, are generated and the results averaged. Table 2 presents these results. The results of the random test suite show that it should be about twenty times larger to achieve nearly full coverage.

Table 2. Results of randomly generated test suite

Tests	Branch Coverage	Code Coverage
400	96.3%	97.80%

During the symbolic execution it was ascertained that one of the radio buttons for choosing company should be pressed for any code to be executed. This demonstrates the applicability of our approach for identifying event sequences that are prerequisite for the execution of a particular path.

3. BACKGROUND

This section provides the reader with some background knowledge about the processes of symbolic execution and GUI testing.

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.

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 (6) is unreachable.

```
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 }
```

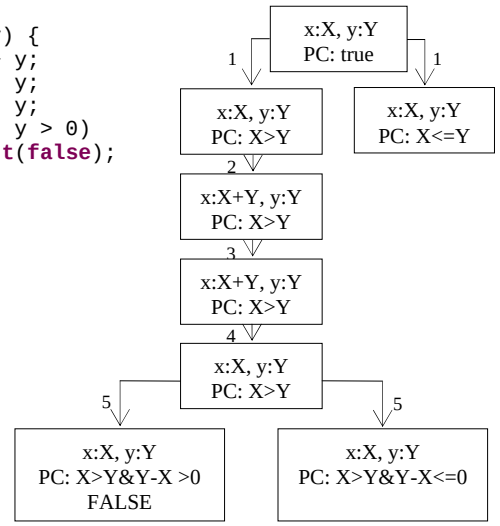

Figure 3. Code that swaps two integers and the corresponding symbolic execution tree

3.2 GUI Event Model

A GUI is an abstraction for providing the user with an interface to interact with an application through actions on virtual objects. User actions trigger events in the GUI. The code executed upon such actions is placed in event handlers that register for a specific event. While the number of possible events accepted by a widget is constant usually not all of these events have corresponding handlers. Hence, focusing the analysis on event handlers restricts the set of GUI events to be explored during the GUI testing process.

3.3 GUI Testing

Since contemporary software extensively uses GUIs to interact with users, verifying GUI's reliability becomes important. There are two approaches to test GUIs.

The first is to keep the GUI light and move all the business logic into the background, thus avoiding the step of GUI testing. In this case 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 techniques. This approach places architectural limitations on system designers.

The second approach is testing to be performed for which several different techniques exist. First, there is the null case of omitting the testing; this leads to production of lower quality software. Second, the GUI is tested with tools that record and replay event sequences [16]. This is laborious and time consuming. Third, tools for automatic test generation, execution, and assessment could be used [7] [10].

Our approach is focused on symbolic execution of event handlers of GUIs. The handlers may implement business logic or delegate to other entities. We instrument the handlers and all methods called from these handlers providing a technique that is applicable regardless where the main processing occurs.

4. BARAD: Symbolic Execution for GUIs

This section presents Barad, our framework for GUI testing. We provide an overview of the processing performed by Barad, the process of symbolic execution, supported symbolic data types, constraint solvers, and test reduction.

4.1 Process Overview

The process of GUI testing performed by Barad is depicted on Figure 4. The initial phase is instrumentation of the GUI event handlers' bytecode using the ASM [2] library. The next step is symbolic execution of the instrumented code.

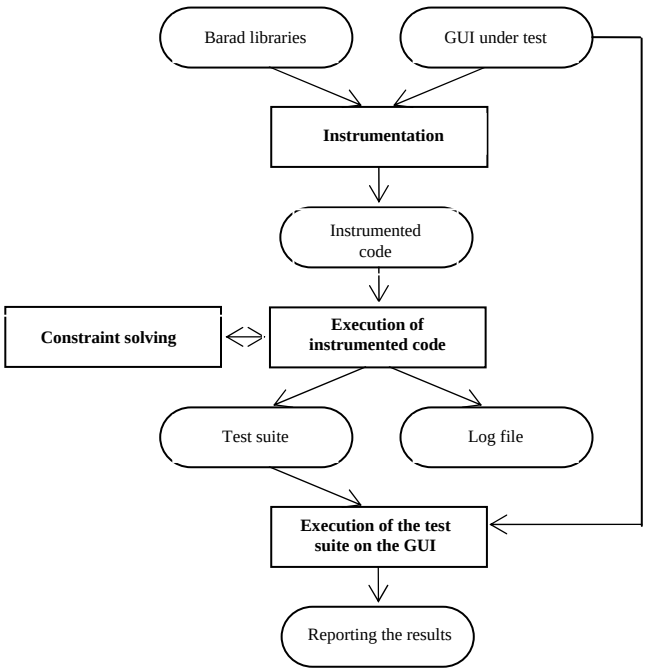

Figure 4. GUI testing process

As a result from the symbolic execution a log file and a test suite are generated. The log file contains constraints on the input variables and concrete values for these variables (if the constraints are satisfiable). The test suite is a script or XML file.

4.2 Symbolic Primitives

Barad provides symbolic equivalents of all primitive types (integer, float, Boolean) and defines the semantic of operations on these types (character is represented as string with length one). For symbolic integers and floats supported operations 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.

4.3 Symbolic Strings

Our symbolic string representation uses finite state automata to model the set of possible values for a string variable. Similar approach is used in [14]. We represent a symbolic string (value of a GUI widget field) as a finite state automaton (provided by [1]) and beginning and end indexes. These indexes define the initial and final position of the automaton sequence that represents the value of the symbolic string. This enables support of operations such as *substring*, *starts with*, *ends with*, and *character at* (in addition to *equal*, *not equal*, and *concatenation*). Path constraints on a symbolic string are used to refine its automaton in way that it rejects values contradicting the accumulated path conditions. The benefit of using automata is that it encapsulates all path conditions and possible values serving as a constraint solver. Note that for each state (created on visiting a new branch of the program) we store a clone of modified variables (required for backtracking) which requires a string constraint solver to aggregate the path conditions on each string variable (in different states) and perform its concretization.

4.4 Constraint Solvers

For solving constraints that arise during the process of symbolic execution Barad requires constraint solvers for numeric data and strings. The architecture of our tool allows the user to specify concrete implementations of the constraint solvers. This is configured in a properties file and a factory dynamically provides constraint solver instances at runtime. Our tool provides default constraint solver implementations for numeric and string data.

The numeric constraint solver uses the Choco [3] library for solving linear constraints on integers and real numbers. It takes as an input symbolic integer entities (variables and constants), float entities, and constrains. The solver is also responsible for generation of concrete values for the input variables (concretization). Note that not all variables passed to the solver are input variables i.e. inputs for the event handlers. Input variables are fields of the symbolic event passed to the event handler or widget field values. During the process of symbolic execution we provide information to the constraint solver which variables are program inputs and should be concretized. Each numeric input variable could define its own range of possible values which is propagated to the solver, otherwise configurable default is used. For infeasible constraints the solver returns an empty set of concrete values.

The string constraint solver takes as input symbolic string entities. Note that constraints on the string entities are captured by their automatons and are not explicitly passed to the solver. The solver is also responsible for concretization of the input variables which are specified during the symbolic execution. In case of infeasible constraints an empty set of concrete values is returned.

4.5 Symbolic Execution in Barad

During the process of symbolic execution we use chronological backtracking to visit all the branches of the program. For each branch we perform the algorithm in Figure 5. First, we create a new state (line 2) which stores the constraint for visiting this branch (line 3) and values of local variables that are to be modified (line 3).

```
1 for (Branch b: branches){
2   Path.createNewState();
3   Path.addConstraint(constr);
4   Path.addVarsBeforeModified();
5   Path.generateTestCase();
6   Path.restoreModifiedVars();
7   Path.removeState ();
8 }
```

Figure 5. State processing algorithm

We store only deltas to the previous state. Note that we instrument Java bytecode and detect modification of a local variable just before that happens via intercepting the bytecode instructions that store values in the local variable table. Before leaving the branch we generate a test case (line 4) and backtrack by restoring the program variables modified in this state (line 6) followed by removing of the state from the state stack (line 7). The state stack is a data structure that stores all the states before the current one. The set of constrains for these states must be satisfied to reach the currently explored branch. Note that branches could be nested and in such cases the state for the outer branch is removed after all nested branches have been explored (i.e. their states removed).

During the bytecode instrumentation we generate an inline version (with branching statements removed) of the program with primitives, strings, and conditional instructions for these types replaced with the corresponding symbolic classes provided by our library. We use as a guide the bytecode labels and jump instructions in the original program to add appropriate bytecode statements that will perform new state creation, backtracking, storing and restoring of local variables, adding path constraints and input variables, reversing path constraints (in the else part of *if*-statements), and logging operations (optional). As a result of this manipulation we obtain an inline, symbolic version of the program which when executed performs symbolic execution of the original code with chronological backtracking. This implementation reaches full branch coverage. Our technique for handling loops is bounded unwinding (configurable).

We perform symbolic execution with inline method code, implement chronological backtracking, and store state deltas because we plan to extend our technique to handle the entire GUI application. In such a case restarting the application (if no backtracking is performed) every time a new path is to be explored could cause significant overhead in terms of execution time.

4.6 Symbolic Execution of Event Handlers

The symbolic execution of GUI event handlers is different from the traditional symbolic execution. First, event handlers have multiple input entry points. Inputs are fields of the event passed to

the event handler (replaced by symbolic event) and the fields of GUI widgets. We identify a widget field as an input variable if its getter is called in the event handler. Second, symbolic execution of event handlers requires uniform handling of symbolic execution over primitives and strings including constraint solving and concretization.

4.7 Test Reduction Algorithm

During the symbolic execution we generate a test case for each visited branch of the program. We adopt such a strategy to identify as early as possible unreachable paths while maximizing code coverage. For example, a branch could be unreachable because the last added path constraint makes the path condition unsatisfiable. Hence, the program path up to the point before adding this path constraint is feasible. By generating test inputs for each branch we guarantee execution of a program path up to the point where its path condition becomes unsatisfiable or a return statement is reached. However, such an approach potentially generates a suboptimal test suite i.e. it is possible an equivalent test suite with less test cases to be constructed.

After generation of a candidate test suite we run the test reduction algorithm presented on Figure 6 that implements three test pruning heuristics. We keep executing the algorithm until no improvement in the test reduction is detected (line 1). Applying one reduction heuristic may enable further refinement by other pruning procedures which did not yield improvement in the previous iteration of the algorithm.

```
1 while (tests.isReduced()) {
2    //remove duplicates
3    for (Test t: tests) {
4      if (tests.isDuplicate(t)) {
5        t.prune();
6      }
7    }
8    //remove subsumed tests
9    for (Test t: tests) {
10     if (tests.isSubsumed(t)) {
11       t.prune();
12     }
13   }
14   //merge disjoint tests
15  for (Test t: tests) {
16    Test disj = tests.getDisjoint(t);
17    if (disj != null) {
18      t.merge(disj);
19      disj.prune();
20    }
21  }
22}
```

Figure 6. Test reduction algorithm

Our first heuristic is simply pruning of duplicate tests. Such tests are generated for disjoint branches in the program with the same path condition. This step is performed first (line 3-7) to avoid subsequent processing of duplicate data.

The second test reduction heuristic is discarding tests subsumed in other tests (line 9-13). Single test may cover several branches in the program for some of which a test may already exist. Consider the example in Figure 7. The messages on line two and six are part of different branches with constraints for printing the message on line two: x > 5, and constraints for printing the message on line six: x > 5 and x > 10. All constraints for executing the code on line two are satisfied by a test which executes the code on line six. In this case our algorithm would discard the test generated for visiting the statement on line two. Currently we are detecting subsumed tests by intercepting duplicate constrains.

```
1 if(x > 5) {
2   System.out.println("x > 5");
3 }
4 if(x > 5)
5   if(x > 10) {
6     System.out.println("x > 10 && x > 10");
7 }
```

Figure 7. Subsumed tests example

The last heuristic we apply to reduce the number of tests is appending compatible tests (line 15-21). Consider the example on Figure 8. There are two input variables: *x* and *y*.

```
1 if(x > 5){
2   System.out.println("x > 5");
3 }
4 if(y < 5) {
5   System.out.println("y < 5");
6 }
```

Figure 8. Disjoint tests example

Lines two and five belong to different branches of the program and visiting each of them depends on disjoint sets of input variables. Since our symbolic execution algorithm generates test inputs for each branch of the program two separate test cases are generated. The first test consists of a concrete value for *x* and the second for *y* respectively. Our algorithm also generates concrete value for each program variable that has been modified in the current branch of the program. This guarantees that test cases with values for disjoint variable sets are safe to be merged since they do not interfere with each other. In such cases we construct a new test case that combines the input values from the merged tests. The only case in which we do not merge tests is if one of the tests is generated for a *terminal* branch i.e. branch containing a return statement.

Once potential tests are reduced a script for their execution on Squish [16] is generated. Barad also writes XML files containing the test suite for each event handler. These files could be used as an input by tools for automatic test execution. Our tool also generates a report containing information about the constraints for each branch, their feasibility, the input variables and their concrete values (if any). In the report unreachable branches are reported as such.

5. CASE STUDY

This section provides a case study using a more sophisticated, application and provides an assessment for the applicability of symbolic execution in GUI testing and the effectiveness of our prototype.

5.1 Subject Application

The application under test is a workout generator used by sports club Apolon. The GUI takes as input user's biometric characteristics such as gender, height, weight, age, metabolism, and experience level.

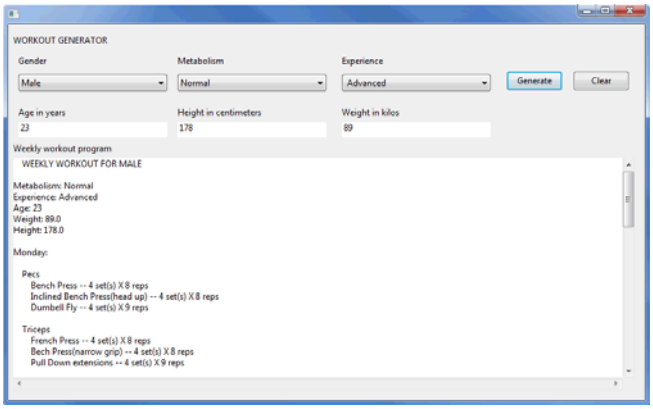

Figure 9. Screenshot of the workout generator

On the basis of this input data the application generates a weekly workout program. Figure 9 shows a screenshot of the Workout Generator GUI.

The application has two event handlers triggered by clicking on each of the buttons respectively. The code first checks if all the user input is provided and if not shows a prompt message to the user. The input widgets consist of three combo-boxes and three text-boxes. Each of the combo-boxes provides an enumeration of possible values: for Gender—*Male*, *Female*; for Metabolism—*Slow*, *Normal*, *Fast*; and for Experience—*Beginner*, *Intermediate*, *Advanced*. These controls are not editable thus always containing a valid input. The text-boxes are initially empty. They accept only numeric characters and for each of them a check for emptiness is performed. The logic of the main generation algorithm has fifty-four branches that depend on values provided by the user. During the process of workout generation, coefficients for the reps, sets and cardio level are adjusted depending on the group to which the user belongs. Depending on the user level of experience different number and kinds of exercises are added to the workout.

5.2 Testing the Subject Application

The Workout Generator is 649 lines of code and has two event handlers. After performing an analysis of the subject application Barad generated a suite of thirty tests. Log file containing control data was also created. Obtained results are presented in Table 3.

The test suite generation time includes bytecode instrumentation, symbolic execution, and test reduction. The full branch and code coverage is not a surprise since symbolic execution generates tests for all reachable branches and all the branches in the event handlers of the Workout Generator are reachable.

Table 3. Results of symbolically generated test suite

Tests	Branch Coverage	Code Coverage	Generation time
30	100%	100%	6.57 sec

Since our approach is focused on generating user inputs and traditional GUI testing techniques rely on manual specification of such inputs, we compare it to random input generation (the general case if a specification is lacking). During random test generation test suites of different sizes are created and run on the GUI. For generation of each test suite, a different seed is used. We generate fifty test suites for each of sizes twenty-five, fifty, one-hundred and two-hundred tests. These tests are generated using the following approach: For combo-boxes, a random choice of value is made. For the age text-box a random value from 10 to 80 is selected. The text-boxes for weight and height accept three digit input. However, there is some realistic upper bound on these biometric characteristics. Upper bound of 220 centimeters for height and 200 kilos for weight are adopted. For concrete value generation the standard Java random generator is used. Notice that this way of random test generation uses some domain knowledge to restrict the number of possible values and differs from pure randomization. Obtained branch coverage from execution of the randomly generated test suites is presented in Figure 10.

While random generation of inputs in its initial phase increases coverage the rate of increase abruptly decreases as covering unvisited branches requires specific values which are unlikely to be generated at random. Table 4 shows the results after execution of a test suite with two-hundred tests. The results are average of fifty runs.

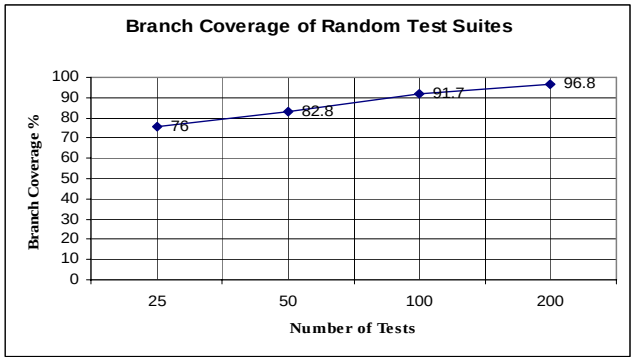

Figure 10. Branch coverage of random test suites

Note that during symbolic execution we have identified an event handler and multiple event sequences (test cases) required to thoroughly test it.

Table 4. Results of randomly generated test suite

Number Of Tests	Branch Coverage	Code Coverage
200	96.8%	99.2%

The randomized approach we used benefits from the fact that we fill all required fields (perform bootstrapping events) before executing the event handler. Traditional GUI testing frameworks are not capable of detecting such a dependency and if a specification is lacking the bootstrapping event sequence would most probably remain unidentified.

6. RELATED WORK

To the best of our knowledge the technique of symbolic execution is not applied in GUI testing. This section investigates the existing approaches for GUI testing, string representation, and some areas where symbolic execution is used for test input generation.

In his Ph.D. Dissertation [7] Memon presents a framework for GUI testing that generates, runs, and assesses GUI tests. This is the first introduced framework capable of performing the whole process of test generation, execution, and result assessment for

GUIs. This framework focuses on the event-flow of GUI applications. For emulating user input a specification based approach is adopted—using values from a prefilled data-base. The main components of the framework are presented in [8], [9], [11], [12]. The most recent research based on this tool is presented in [18] by Memon and Xie. Our approach focuses the data-flow of GUIs and is complementary to this work.

Memon, Banarjee and Nagarajan present a framework for regression testing of nightly/daily builds of GUI applications [10]. This tool addresses the rapidly evolving GUI applications executing small enough test suite that the test process could be accomplished in less than a day/night. This tool also uses a specification based approach for emulating user inputs and is complementary to Barad.

Another approach is the GUI to be represented as a Variable Finite State Machine from which after a transformation to an FSM, tests are obtained [15]. This approach does not consider user input while focusing on the event-flow of GUIs. Such a technique could be adopted in Barad to enable the capability for complete testing of GUI applications.

A technique that transforms GUIs into a FSM and uses different techniques to reduce the states of the FSM to avoid state space explosion is proposed in [17]. In this work the focus is on collaborating selections and user sequences over different objects in the GUI. User input is not handled and again this technique is complementary rather than competing to our approach.

Symbolic execution for test data generation is used in [19]. The program is represented as a deterministic FSM and using symbolic execution test data is generated. This work deals exclusively with numeric constraints. Barad performs symbolic execution over strings (in addition to primitives). Also the input variables for GUI event handlers have multiple entry points opposed to this approach where input variables have a single entry point—the method parameters.

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 non-conflicting with the path conditions values.

7. DISCUSSION

In this section we discuss some limitations of our approach which are inherent from the technique of symbolic execution and our concrete implementation.

Loop handling during symbolic execution is a bounded unwinding i.e. the code in the loop is executed iteratively up to a certain bound (specified by a property in the Barad configuration file). It is possible that some path of the program remain unexplored if the number of loop unwindings is lower than the iterations required for modifying a program variable (used latter on in a conditional statement) to a value required for visiting a particular branch. Another limitation of the symbolic execution is constraint solving. Our numeric constraint solver handles only linear constraints (Anecdotal evidence suggests that most of the program path constrains are linear) while during the symbolic execution non-linear constrains may arise. Further, not all path constraints are decidable. In such cases some program paths would remain unexplored. Further, we support a limited set of operations on symbolic string entities (presented in Section 4.3). In addition, our string constraint solving does not handle regular expressions.

Our current implementation performs Java bytecode instrumentation which restricts the potential test subjects to applications written in the Java programming language. Our technique could also be applied to applications developed with the .NET framework by instrumentation of the Microsoft Intermediate Language (MIL).

During the GUI testing process we define a test case as failed if an uncaught exception is thrown. Such a test oracle could potential identify program faults but does not allow checking of richer GUI properties. We are currently working on techniques for introducing stronger test oracles.

8. CONCLUSION

We introduced a technique for systematically checking GUI applications by symbolically executing the event handlers of Java applications. Our tool Barad performs automatic Java bytecode instrumentation and concrete input generation for the data widgets of the GUIs.

Experimental results using our prototype show that it provides significantly better performance compared to random input generation, in terms of line and branch coverage. Barad also captures event sequences that transform a GUI in a state appropriate for execution of a particular segment of event handler code.

Barad complements the traditional approaches for GUI testing by providing a technique for testing a class of GUI applications that conventional approaches could not effectively verify. We believe that combining our approach with existing frameworks [7] [10] presents a promising approach for systematic testing of GUIs.

9. REFERENCES

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

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

[3] Choco, Retrieved on January 25, 2008 from ASM: http://choco-solver.net/index.php?title=Main_Page

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

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

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

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

[8] 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. (BibTeX).

[9] 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.

[10] 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. (BibTeX).

[11] 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. (BibTeX).

[12] 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.

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

[14] 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.

[15] 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 (FTCS '97)*, p. 80, 1997.

[16] Squish, Retrieved on January 25, 2008 from FrogLogic http://www.froglogic.com/pg?id=Products&category=squish&sub=overview&subsub=overview

[17] White, L., and Almezen, H. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *11th International Symposium on Software Reliability Engineering (ISSRE'00)*, p.110, 2000.

[18] 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

[19] 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

[17] White, L., and Almezen, H. Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In *11th International Symposium on Software Reliability Engineering (ISSRE'00)*, p.110, 2000.

[18] 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

[19] 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