Symbolic Execution for GUI Testing

Svetoslav Ganov
Laboratory of Experimental Software Engineering
University of Texas at Austin
001-832-366-4884
svetoslavganov@mail.utexas.edu

Sarfraz Khurshid
Software Testing and Verification Group
University of Texas at Austin
001-512- 232-7927
khurshid@ece.utexas.edu

Dewayne Perry
Laboratory of Experimental Software Engineering
University of Texas at Austin
001-512- 232-3343
perry@ece.utexas.edu

ABSTRACT

A Graphical User Interface (GUI) is an abstraction providing users with a more natural way of interacting with computers. It consists of objects like buttons, text boxes, toolbars etc. The communication between users and GUIs is event driven. Users can modify the state of a GUI and trigger events that lead to the execution of different code fragments. Hence, in order to test a GUI one should execute event sequences simulating user behaviors. While the state of some GUI widgets is limited to a small number of values (the value of a radio button), others have a wide range of possible states (the value of a text box). Such widgets are used for data input from the user in the form of text (alphabetic or numeric). Since program execution may depend on the user input, it is a challenge to select suitable values in a way that allows thorough testing. We propose symbolic execution for obtaining these inputs. During symbolic execution, each branch of the program is visited and the constraints for control variables are resolved determining if it is reachable or not. Thus, by symbolically executing code that depends on user input, we can obtain values that ensure visiting each reachable branch in the program.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Symbolic execution, Testing tools*

General Terms

Verification.

Keywords

Software testing, GUI testing, Symbolic execution

1. INTRODUCTION

A Graphical User Interface (GUI) provides the user with a more convenient and intuitive way for interaction with the computer. It was first introduced by Dough Englebart, an employee in Stanford Research Institute in 1965, and the first publicly affordable computer with a GUI— the Macintosh—appeared in 1984 [17]. A GUI is a set of virtual objects (widgets) that are more intuitive to use, for example buttons, edit boxes, etc. In other words, it is a way of facilitating communication between humans with their complex vision of the world and computers and their software systems. GUIs are ubiquitous. Almost every computer user takes advantage of the convenience provided by that abstraction. 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. If one wants to test a form with five buttons (a button being the simplest active GUI widget), he must try all the 120 possible combinations. This is necessary because in the internal logic of the GUI triggering of one event before another may cause the execution of different code segments. This limitation relates to the black-box testing of GUIs. Further, there are some GUI widgets that are used for user input (text boxes, edit boxes, combo boxes etc.) and could have an extremely large space of possible inputs. However, their content might cause some branching in the program. The most trivial approach in this case could be choosing inputs in a random fashion, but the large space of possible values makes it very likely that some parts of the underlying code would remain unexecuted. It is enough for one to consider the possible values for a ten character word and a program that executes a piece of code for a particular value of that input. It is almost certain that in this case randomization would not find that "special" value.

A specification-based (black-box) approach may find such inputs, however it would require detailed specifications, which are often not feasible to write. We propose a white-box testing approach for identifying these values: symbolically execute the underlying GUI code and generate a test suite that gives high (ideally full) coverage.

Our approach is particularly suited for testing GUI applications that take user input in the form of text as parameters and have complex branching behavior that depends on the values of these parameters. Existing GUI testing methodologies are not effective at testing such GUIs as the focus of traditional work on GUI testing has been on checking properties of event sequences and not data dependent behavior [8] [11] [15] [16]. Data is typically abstracted using a small set of expected values. Since the focus of our technique is on data, it is complementary to the existing approaches and leads to an increased branch and code coverage. Moreover, our approach enables a reduction in the number of tests needed to systematically check a GUI, since it inherits the strengths of symbolic execution, which enables exploring different program paths systematically and (for decidable constraints) detecting infeasible paths. We have implemented a prototype Barad for testing GUI applications developed in C#.

We make the following contributions:

- **Symbolic execution for GUI testing.** We introduce the idea of systematically testing GUI applications using symbolic execution.
- **Algorithm.** We present an algorithm for systematic testing of GUIs; the algorithm implements an efficient solver for constraints on primitives and strings; it also minimizes generated test suites.
- **Implementation.** Our prototype Barad implements our algorithm for testing C# applications.
- **Evaluation.** We evaluate our approach using GUI subjects inspired by commercial applications.

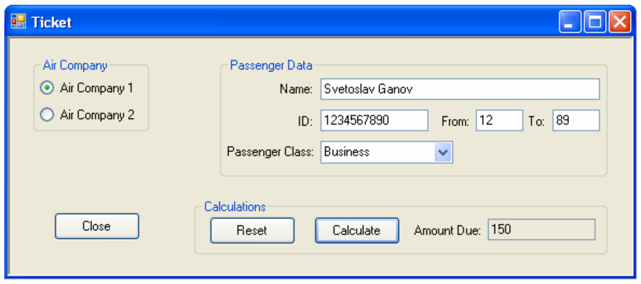
This paper is organized as follows. Section 2 gives an overview. Section 3 provides some background information. Section 4 presents Barad. Section 5 is a case study. Sections 6, 7, 8 give some related work, future work and conclusions, respectively.

2. OVERVIEW

This section provides the basics of how our technique is applied to GUI testing. The goal is to give an overview of the methodology and show an example where conventional GUI testing techniques would fail to achieve high coverage unless a prohibitively large test suite were used.

2.1 Example application

The example GUI presented in Figure 1 is specially designed for the current research. It is inspired by an industrial application that the first author developed in C# during his past work. It is a simple GUI program that calculates the amount due for a plane ticket depending on the distance, passenger class, and selected airline.



Name: Svetoslav Ganov
ID: 1234567890 From: 12 To: 89
Passenger Class: Business
Amount Due: 150

Figure 1. Example application

The main calculation is performed after the button “Calculate” is pressed. In order for some calculation to be done the user must choose at least one company and enter his name and ID. There are different passenger classes—namely—“Business”, “Child”, “Student” and “Privileged”. The first three categories are offered in a combo box menu and the last one should be typed explicitly. (Note that only when the user enters this special value manually will the corresponding code be executed) Moreover, if the “Passenger class” contains a value that is not in the above enumeration an exception is thrown. Each company has its own coefficient that is used during the calculation. Further, the different passenger classes have different base prices depending on the distance to be traveled, which is the absolute difference between the value in the boxes “From” and “To”. The main goal was to create an application with several branches the execution of which depends on user input both in the form of text and event sequence (i.e. selecting a radio button).

Figure 2 shows a code fragment for the “Business” class. The source code for the other passenger classes is similar. As one can see there are five different branches that may be visited if the traveled distance is in a definite range. The calculation method has 22 branches plus one for the exception thrown if the “Passenger class” value is unacceptable. The conditions are nested three levels. In order to visit all of these branches, one must select a company, a passenger class, and enter at least one combination of departure and destination codes so that their absolute difference satisfies the specified branch condition. Moreover, to reach that code the user should have been entered his name and ID.

```
{
 case "Business":
 {
  if (0 <= distanceRange && distanceRange < 50)
   amountDue = 120 * coeficient;
  if (50 <= distanceRange && distanceRange < 60)
   amountDue = 130 * coeficient;
  if (60 <= distanceRange && distanceRange < 70)
   amountDue = 145 * coeficient;
  if (70 <= distanceRange && distanceRange < 80)
   amountDue = 150 * coeficient;
  if (80 <= distanceRange && distanceRange < 100)
   amountDue = 160 * coeficient;
  break;
 }
```

Figure 2. Code fragment for the “Business” class

Our technique performs the following steps: Tested code is instrumented using the symbolic data type system. Follows execution of that code during which control text files and a test suite are generated. The last phase is execution of the test suite on the application. Figure 3 depicts the flow of this process.
During instrumentation, two simple rules are followed: 1) variables and operations on them are replaced by the corresponding symbolic types; 2) the branching constructs are substituted by the respective chooser classes. Section 5 presents a case study and examples of instrumented code. The generated test suite is a C# source file. To execute the tests we open the source file in Visual Studio, compile it, and run it.

During test execution on the example application one of the tests fails because it attempts to explore a branch where the input for passenger class is not recognized. In this way the branch that throws an exception in case of unacceptable input is explored. Branch coverage, code coverage and execution time for this test suite are presented in Table 1.

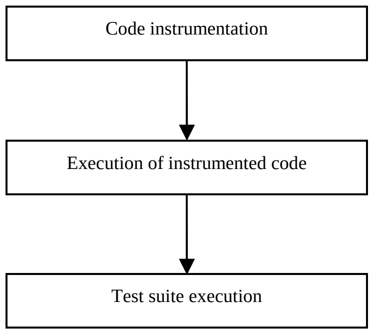


Figure 3. Test process flow

Table 1. Results of symbolically generated test suite

Number Of Tests	Branch Coverage	Line Coverage	Execution Time
23	100%	100%	4.92 sec

These results are not surprising since all the branches of the tested code are reachable.

We next compare our approach to random test generation. Our methodology to create a random test suite is as follows. During the symbolic execution it was ascertained that the execution of the code depends on the selection of a company (clicking on a radio button). The randomization for that case is modeled by selecting one of the three options—Air Company1, Air Company2, and neither. Since there are three possible values for the passenger class plus one that should be typed explicitly, a random choice is made out of the following values—one of the passenger types in the enumeration, the one that should be typed explicitly and just leaving the control empty, which is equivalent to incorrect input. The input for the text boxes "From" and "To" is a number between 0 and 99 and is chosen randomly. For the randomization, the C# random class is used. In order to provide more accurate results fifty test suites, each with different seed, are generated and the results averaged. Table 2 presents these results.

Table 2. Results of randomly generated test suite

Number Of Tests	Branch Coverage	Line Coverage	Execution Time
400	97.1%	98.86%	46.17 sec

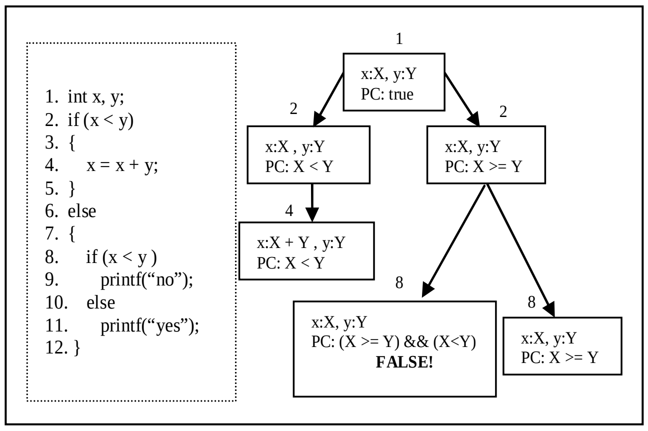
The results of the random test suite show that it should be about twenty times larger in order to achieve high coverage. This explains the significant difference of the execution time. The number of tests is ascertained with running smaller test suites and intermittently increasing the number of tests until almost full coverage is achieved. Results in Table 2 are to be interpreted as follows: since 4/5 of the input for passenger class is valid, the absolute difference between departure and destination, and a radio button selection are primarily responsible for visiting a particular branch. As can be seen from this example, during the symbolic execution it is ascertained that one of the radio buttons for choosing company should be pressed for a particular code segment to be executed. This further proves the applicability of the symbolic execution not only for generation of user inputs but also for modification of the GUI state in a manner suitable for reaching a particular branch.

3. BACKGROUD

This section provides the reader with an overview about the processes of symbolic execution and GUI testing.

3.1 Symbolic Execution

The process of symbolic execution is actually execution of the original program, but instead of concrete values its variables are symbolic. Hence, every time a variable is modified, its current value becomes a function of its initial symbolic state and the accumulated operations up to that moment. During symbolic execution the program has different values for each symbolic variable, a program counter, and a path condition. The path condition contains accumulated constraints over the input variables that must be satisfied in order for this branch to be executed. It is a quantifier-free Boolean formula, such that if evaluated as true then there is at least one combination of input values that will execute the current branch. Intuitively, if the path condition evaluates to false that means no input satisfies that expression and this branch is unreachable. Every time the program encounters a branch statement, all the possible outcomes must be taken into account. Hence, if execution reaches an "if" statement, there are two possible scenarios that must be considered. This exploration of all possible paths forms the "execution tree" of the program with nodes representing the possible states and edges representing state transitions. Consider the example in Figure 4. The initial values of the variables "x" and "y" are set to symbolic ones. Every time a symbolic variable is modified it accumulates the update as an expression. When a branch is reached all the possible paths are explored and after entering a branch the path condition (PC) is updated and its validity checked.



```
1.  int x, y;
2.  if (x < y)
3.  {
4.      x = x + y;
5.  }
6.  else
7.  {
8.      if (x < y )
9.          printf("no");
10.     else
11.         printf("yes");
12. }
```

Figure 4. Program and its execution tree.

3.2 GUI Testing

Since most contemporary software uses GUIs to interact with users, verifying a GUI's reliability becomes important. In general, there are two distinct approaches to testing GUIs. The first keeps the GUI as light-weight as possible and move all the business logic into the background, thus avoiding this 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 software testing techniques. This approach however, places architectural limitations on system designers. The second uses some sort of GUI testing to verify its correctness. There are several different ways for one to accomplish this. First, there is the null case of omitting the GUI testing; this leads to production of lower quality software. Second, the GUI is tested with tools that can record and replay event sequences performed by the tester such as the one presented in [18]. This technique is laborious and time consuming. Hence, it is difficult for the tester to come up with a large test suite. Another approach is to use tools for automatic test generation, execution and assessment like the ones presented in [8] and [11]. These appear to be the only such complete frameworks and they also have the same author.

All mentioned techniques, however, (aside from omitting testing) have a common drawback concerning the user input. They do not propose any mechanism of obtaining input values that would affect the program flow and lead to execution of different code sections [8] [11] [15] [16]. In manual test generation the tester must be aware of such “special” inputs in order to create relevant tests. The same is true for the automatic testing where during test generation values are read from a filled in advance database [8] [11].

4. BARAD: Symbolic Execution for GUI Testing

In general there are two different ways for one to do symbolic execution. First one is to use a model checker, since contemporary model checkers can perform symbolic execution. In this case the tested code must be either translated into the language of the model checker (construction of a model) or it should be instrumented and used as input for a model checker that supports the implementation language. This approach is used in [5] where Java programs are instrumented and run with the Java PathFinder model checker [20]. A second approach for implementing symbolic execution is to create symbolic system of classes representing each corresponding basic data types (integer, string, Boolean etc.) and to define the semantics of the operations on them.

Since the goal of the current project was to explore the applicability of symbolic execution for GUI testing in C#, the only available model checker for that language was Zing [1]. Zing is still in development and does not support symbolic operations on strings. The goal of this research was to obtain interesting input values for GUI testing, and this input, in general, is in the form of strings for both alphabetical and numeric data. If there is a transformation from strings to numbers, it is done explicitly in the code. Of course, for string representations in a model checker, an abstraction could be used; however, operations like “substring” are still challenging.

For the purpose of this research, a framework for symbolic execution in C# was implemented. It is called Barad and deals, with some limitation, with all basic data types, namely integer, numeric (double), Boolean, character, and string. Barad does not yet support automatic instrumentation, which is performed manually in a way to guarantee the execution of each branch of the program and is the only time consuming manual step in the process.

The next paragraphs give an overview of the data type system implementation, the semantics on supported data types, the constraint resolving techniques, and the test case generation algorithm.

4.1 Data types

All the basic data types in Barad implement a common interface that defines auxiliary methods used during the execution of a symbolic model. There are three main types of integer entities—constant, variable, and operation. The following operations are currently supported for integer expressions—and, or, addition, difference, multiplication, division, less than, greater than, greater that or equal, less than or equal. Each integer operation is a class that inherits the class for integer operation which implements an integer interface. The integer interface on the other hand inherits the common interface. This results in a class system capable of representing symbolic operations over integers and allows their unbounded nesting. Boolean data type is not explicitly implemented, since it is straightforward to use integers instead of Booleans. Doubles are represented in a similar way to integers. Supported operations are: and, or, addition, difference, multiplication, division, less than, greater than, greater than or equal, less than or equal. Strings are represented by analogous class system. Currently implemented operations are equal, not equal, concatenation, and substring. The character type could be viewed as string with length one.

During symbolic execution, the constraints over input variables should be resolved when entering a new branch. Hence, a technique for resolving these constraints is needed.

4.2 Constraint solving

For resolving of numeric constraints, the previous version of Barad used a theorem prover CVC3 [21] capable of evaluating complex expressions. However, CVC3 evaluates formulas in a given logical context declaring them as valid or invalid without providing concrete assignments to the input variables. This is a serious drawback since we aim to generate input values for the GUI. Because of that limitation a constraint solver is implemented in Barad. It can handle relatively simple constraints for integers and doubles and uses set restriction to evaluate the range of possible values for each variable. This technique does not handle the general case but solves all numeric constraints in the tested GUIs. Currently under development is a constraint solver that uses linear optimization (simplex method) for solving constraints and generation of concrete inputs. It will recognize the simple case in which all constraints are or could be represented as difference constraints, thus solving them and generating concrete values could be cast to a shortest path problem.

We use the idea that each GUI widget used for keyboard input has a maximal length for solving string constraints. Each string variable or constant is represented as a bounded set of characters, each of which could take a bounded set of possible values. The difference between a variable and a constant is that the constant has only one possible value for each character. Equality is represented as intersection of the corresponding sets while inequality is represented as the exclusion of that intersection. Concatenation of strings is concatenation of the representation sets. Substring is a subsection of the corresponding set. Since only the allowed values for a character are stored in these sets, a random number generator is used for choosing a value iterating for every position of the string. However, this technique has a limitation. Consider the following example. A two character string variable *S* is represented as a set of sets with possible values and length two. If one imposes a constraint that *S!="ab"* characters *"a"* and *"b"* are removed from the corresponding sets. Now, if one adds a new constraint that *S="aa"* the condition would be unfeasible because "a" is no longer present in the first set. However, this representation works fine for the values and operations found in the tested applications.

Moreover, Java String Analyzer [2] performs static analysis of Java programs and generates a context-free grammar for each string expression represented as a multilevel automaton. Unfortunately, this tool works only on Java programs. However, this technique could also be implemented in C# since only the front end of the tool is language dependant.

In some cases a string variable is used to represent the value of a combo box implemented as a drop down list that can take only a finite set of possible values. In such case the variable contains an enumeration of the possible values. When an enumeration is provided it has precedence over the set representation.

Note that the main contribution of this paper is the idea of using symbolic execution for GUI testing, rather than offering general solutions for string representation and constraint solving. Implemented solvers are effective enough for evaluating all the constraints in the tested applications. Therefore, we could successfully assess the applicability of symbolic execution for GUI testing.

4.3 DFS traversal and backtracking

The technique of symbolic execution imposes visiting of all branches of the program. This is performed by two classes—"ChooseBool" and "ChoosePath"—which are used during the instrumentation phase to replace the standard branching constructs—"if" and "switch", respectively. (The case study section contains examples of instrumented code using these classes) These objects choose non-deterministically all possible paths in a loop until all paths are explored. This implementation provides a DFS traversal of the program execution tree.

Path condition is implemented as a separate class. It accumulates constraints over input variables and generates tests. Path condition is subscribed for the new branch event generated by each path chooser and every time a new path is selected the event handler executes an algorithm performing the following steps:

1. Accumulated constraints are evaluated.
2. If all constraints evaluate to true and are not contained in any existing test record, a new test record is created, and values for the input variables are generated.
3. If at least one constraint evaluates to false, a test record is created (containing only constraints without input values)
4. Unneeded constraints are removed
5. Program variables are restored

Constraints are stored in a dynamic array (ArrayList object). Before a new branch is taken by the path choosers, an auxiliary object that has twofold purpose is added to the array. First, it serves as a separator of the constraints added by different choosers containing the unique ID of the chooser that created it. Second, it stores values of the program variables before entering the new branch. When a new branch event is generated all constraints and constraint dividers beginning form the end are removed consecutively until a constraint divider with ID equal to the event sender is reached. Then all program variables are updated with values stored in the constraint divider and it is removed from the array list.

4.4 Test case generation

The initial idea was for each leaf operation of the execution tree a separate test case to be generated. However, this approach does not provide the minimal number of tests. Moreover, if there are several branches of the program with the same condition there would be doubling of tests which is undesirable. Further, it is possible that one test covers several leaf operations. Consider the example in Figure 5.

```
1 if (x > 5)
2  MessageBox.Show("x > 5");
3
4 if (x > 5)
5  if (x > 10)
6    MessageBox.Show("x > 10 && x > 10");
```

Figure 5. Overlapping tests example.

The message boxes on line 2 and 6 are part of different leafs of the execution tree. Hence, a constraint for message box on line 2 to be shown is: x > 5 and constraints for showing the one on line 6 are: x > 5 and x > 10; However, all the constraints for executing the code on line 2 are included in the constraints for execution of the one on line 6. Hence, we don't need to generate two separate test cases. Rather, the one for executing line 6 would guarantee execution of the code on line 2. In this way we reduce the number of tests.

Another possibility for test reduction is the merging of compatible tests. For example, let us have two input variables and two branches the execution of which depends only on one of these variables, respectively. These branches correspond to different leafs of the execution tree and their constraints are not contained in one another. Hence, there will be two separate test cases generated. It is possible these two tests can be merged. We use a simple heuristic to detect when we can merge tests. There are two conditions that have to be met: First, neither variable should be modified inside the branches determined by the other one. Second, this techniques cannot be applied if one of the branches in the execution tree is "terminal" i.e. contains a return statement. We verify that there is no reduction in coverage. If there is such we do not merge the tests.

During the symbolic execution, potential test cases are stored in a data structure. Every time a new branch is executed, if all of its constraints are valid, these constraints are checked to see if they are a subset of the ones belonging to any of the stored potential tests. If true, no test object is instantiated. However, this guarantees only that every potential test is not covered by previously generated ones and there is no guarantee that previously generated tests are not covered by the current one. Therefore, after completion of symbolic execution the data structure containing potential tests is traversed. During the first traversal all the overlapping test cases are dropped using the first reduction heuristic. If the option for merger of test cases is set to true another traversal is made and compatible tests are merged using the second reduction heuristic.

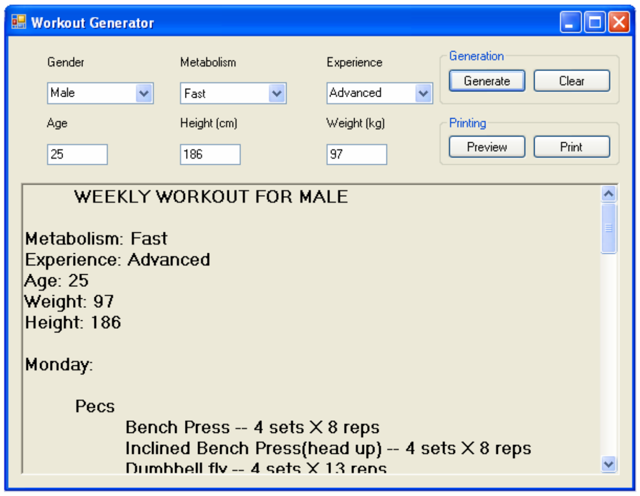
Once potential tests are reduced a C# source file containing the test suite is generated. Moreover, for each potential test a text file is created containing information of the constraints, their feasibility, variables and concrete values, if any. Note that for each unreachable branch a potential test case is generated, even though it is not considered during the test reduction phase. Since it is unreachable no test is added to the test suite and only a text file containing the corresponding constraints is created. This is useful for identifying and fixing unreachable branches.

5. CASE STUDY

This section provides a case study using a more complicated, real application and provides a preliminary assessment for the applicability of symbolic execution in GUI testing.

5.1 Application under test

The application under test is a workout generator used by the members of sports club Apolon. This automation facilitates clients in designing workout that fits their needs, speeds up the process and leads to economies reducing the number of personal instructors. The program takes as input user's personal biometric characteristics such as gender, height, weight, age, metabolism and his or her experience level. On the basis of this input data the application generates a suitable week workout program. Figure 6 shows a screenshot of the Workout Generator GUI.



Gender: Male
Metabolism: Fast
Experience: Advanced
Age: 25
Height (cm): 186
Weight (kg): 97

```
        WEEKLY WORKOUT FOR MALE

Metabolism: Fast
Experience: Advanced
Age: 25
Weight: 97
Height: 186

Monday:

        Pecs
                Bench Press -- 4 sets X 8 reps
                Inclined Bench Press(head up) -- 4 sets X 8 reps
                Dumbbell fly -- 4 sets X 13 reps
```

Figure 6. Screenshot of the workout generator

The application has several event handlers corresponding to clicking on each of the buttons. All of them except the one for clicking the "Generate" button do not perform any calculations, but rather have auxiliary functions as resetting the form, printing the workout etc. Since the current implementation of Barad still does not support automatic code instrumentation this process has to be done by hand. The code itself first checks if all the user input is provided and if not it shows a message urging the user to enter the required data.

The input widgets consist of three drop-down lists and three text boxes. Each of the drop-down lists provides enumeration of possible values: for "Gender" are "Male" and "Female"; for "Metabolism" are "Slow", "Normal" and "Fast"; and for "Experience" are "Beginner", "Intermediate" and "Advanced". These controls do not allow other way of interaction than selection of an item from the list and are properly initialized. Hence, it is not possible for the user not to provide values for these widgets.

However, the text boxes are initially empty and require input of corresponding values. They accept only numeric characters and for each of them a check if it is empty is performed. The logic of the main generation algorithm has fifty-four branches that depend on values provided by the user. During results generation, coefficients for the reps, sets and cardio level are adjusted depending on the group to which the user belongs. Further, depending on the user level of experience different number and kinds of exercises are added to the workout.

5.2 Results

The length of code that is instrumented is 460 lines and has 54 branches. Figure 7 shows an example of instrumented "if" statement using the path chooser ChooseBool. The object cb11 (line 1) chooses non-deterministically (enumerates) one of the two possible paths corresponding to the "if" and the "else" statement of the construct. On line 6 the constraint for visiting the particular branch is added to the path condition. Lines 8-15 are equivalent to:

rtBox1 += "string value";

rtBox1 += "string value";

In this case there is no "else" statement in the "if" construct.

```
1  ChooseBool cb11 = new ChooseBool(pc);
2  while (!cb11.allExplored())
3  {
4   if (cb11.nextPath())
5   {
6    pc.AddConstraint(experience.SEQ("Advanced"));
7
8    rtBox1.CONCAT("\t\tInclined Bench Press(head up) –
9            " + setCoef.getStrValue() + " sets X " 10  +
10           repCoef.IADD(5).getStrValue() + " reps\n");
12
13   rtBox1.CONCAT("\t\tDumbel fly -- " +
14           setCoef.getStrValue() + " sets X " +
15           repCoef.IADD(5).getStrValue() +"reps\n");
16   }
17 }
```

Figure 7. Instrumented code example

Figure 8 shows an example of an instrumented "switch" statement using the path chooser ChoosePath. The object cp5 (line 1) receives as a parameter the number of branches that are to be explored and the path condition, which registers for receiving the new branch event of cp5. cp5 chooses branches non-deterministically—it just enumerates all integer values from one up to the initialization value minus one. This value is used for conditional switch statement (line 4). On lines 8, 14, 20 and 26, belonging to different branches of the switch statement, constrains are added to the path condition. On lines 9, 15, 21, and 27 calculations adjusting the cardio coefficient are performed and have the following equivalent:

coefficient = coefficient * double constant

```
ChoosePath cp5 = new ChoosePath(4, pc);
while (!cp5.allExplored())
{
 switch 4(cp5.nextPath())
 {
  case 0:
  {
   pc.AddConstraint(age.IGTEQ(0).IAND(age.ILTEQ(20)));
   cardCoef.SetOp(cardCoef.DMUL(0.5f));
   break;
  }
  case 1:
  {
   pc.AddConstraint(age.IGT(20).IAND(age.ILTEQ(30)));
   cardCoef.SetOp(cardCoef_expr.DMUL(1.2f));
   break;
  }
  case 2:
  {
   pc.AddConstraint(age.IGT(30).IAND(age.ILTEQ(45)));
   cardCoef.SetOp(cardCoef.DMUL(2.3f));
   break;
  }
  case 3:
  {
   pc.AddConstraint(age.IGT(45));
   cardCoef.SetOp(cardCoef.DMUL(1.6f));
   break;
  }
 }
}
```

Figure 8. Instrumented code example

The rest of the code is instrumented similarly. After successful instrumentation, the code is run and a test suite of thirty tests is generated. Text files containing control data are also created. A sample of such file containing the constraints and values for input variables, if constraints are feasible, is presented in Figure 9.

```
                    TestCase30
Constraint:  (Convert.ToInt16(textBox1.Text) > 0) ; Valid: True;
Constraint:  (Convert.ToDouble(textBox2.Text) > 0) ; Valid: True;
Constraint:  (Convert.ToDouble(textBox3.Text) > 0) ; Valid: True;
Constraint:  (comboBox3.Text = "Male") ; Valid: True;
Constraint:  (comboBox2.Text = "Beginner") ; Valid: True;
Constraint:  ( (Convert.ToInt16(textBox1.Text) >= 0)  AND
             (Convert.ToInt16(textBox1.Text) <= 20) ) ; Valid: True;

Variable: Convert.ToInt16(textBox1.Text); Value: 11;
Variable: Convert.ToDouble(textBox2.Text); Value: 245;
Variable: Convert.ToDouble(textBox3.Text); Value: 27;
Variable: comboBox3.Text; Value: Male;
Variable: comboBox2.Text; Value: Beginner;
```

Figure 9. Control text file example

During test case generation a library provided by NuntitForms [18], a new record/replay tool for testing C# GUIs is used. Figure 10 shows an example of a generated unit test. In order to test a GUI, an instance of the form has to be instantiated and run (lines 3, 4). Further, for each widget of the GUI an object of the corresponding tester class must be created (lines 6-15). For example, the class ButtonTester is used for testing a button control and takes as a parameter for its constructor the name of a button (line 6).

```
1 public void Test8()
2 {
3  WorkoutGenerator formToBeTested=new WorkoutGenerator()
4  formToBeTested.Show();
5
6  ButtonTester button1 = new ButtonTester("button1");
7  ButtonTester button2 = new ButtonTester("button2");
8  ButtonTester button3 = new ButtonTester("button3");
9  ButtonTester button4 = new ButtonTester("button4");
10 TextBoxTester textBox1 = new TextBoxTester("textBox1");
11 TextBoxTester textBox2 = new TextBoxTester("textBox2");
12 TextBoxTester textBox3 = new TextBoxTester("textBox3");
13 ComboBoxTester comboBox1=newComboBoxTester("comboBox1")
14 ComboBoxTester comboBox2=newComboBoxTester("comboBox2")
15 ComboBoxTester comboBox3=newComboBoxTester("comboBox3")
16
17 textBox1.Enter("38");
18 textBox2.Enter("53");
19 textBox3.Enter("194");
20 comboBox2.Select(2);
21 comboBox3.Select(1);
22 button1.Click();
23 }
```

Figure 10. Generate test case

The next step is to execute a sequence of events on the GUI such as clicking a button, inputting text, and so forth (lines 17-22). Once created the test suite is run on the GUI with an add-in for Visual Studio called TestDriven.net [19]. This tool executes tests in a separate process that is kept alive in case other tests are to be run. Is also provides detailed information about the line coverage attained during testing. The created test suite consists of thirty tests and after its execution the results in Table 3 are obtained.

Table 3. Results of symbolically generated test suite

Number Of Tests	Branch Coverage	Line Coverage	Execution Time
30	100%	100%	4.35 sec

The results from execution of the test suite derived by symbolic execution achieve maximal branch and line coverage. This is not a surprise since generated tests guarantee execution of each feasible branch and all the branches of the instrumented code are reachable. Thus, a test suite obtained by executing symbolically the code under test gives the tester confidence in the thoroughness of performed testing.

However, to assess the advantages of the proposed technique it should be compared to other approach for testing the GUI. To the best of our knowledge all other work in GUI testing is focused on event sequences and does not provide any mechanism for generating user data inputs. These techniques are black box testing approaches and abstract the GUI as a graph, FSM etc [8] [11] [15] [16]. The problem with the user input is either not considered [15] [16] or it is looked-up in a database [8] [11]. However, the question that arises is how to fill that database. In general, there are two possible approaches: perform some sort of code analysis or generate random values. Since the technique proposed in this paper generates inputs by code analysis and to the best of our knowledge there is no other such one applied in GUI testing, it should be compared to random input generation. Such a comparison should provide answers to the following question: Is the proposed approach better? How much better is this technique than the alternatives?

Test suites of different sizes are created using randomization and run on the GUI. For generation of each test suite, a different random 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 drop down list (combo boxes), a random choice of value is made. The same is done for the text boxes indicating age, height and weight. Since there are no widely accepted lower or upper bounds for age in order to workout, and the maximal input length is two digits, a value from the whole range 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 C# random generator is used. Notice that this way of random test generation uses some domain knowledge to restrict the number of possible values and this differs from pure randomization, thus increasing the effectiveness of generated test suite.

The main observed testing criterion is branch coverage, since it is the most realistic indicator for the effectiveness of testing. Even though, there is a correlation between line and branch coverage, it is not guaranteed that almost full line coverage would result in nearly the same branch coverage. The reason for that is that length of the code in different branches may vary significantly. However, if one achieves full branch coverage, the full line coverage is guaranteed. Obtained results are presented in Figure 11. As it is well known random test generation is quite successful in the beginning but as one comes closer to full coverage, progress gets much slower. Note that the results of this study are congruent with that fact. The first twenty five tests achieve 76% branch coverage but after doubling the size of the test suite branch coverage increases only with 6.8% up to 82.8. Our data show that for almost full branch coverage a test suite with approximately 200 tests is needed. Compared to the test suite obtained by symbolic execution this is almost seven times larger. Moreover, it does not guarantee that all branches of the program are visited. It is possible that exactly the missed small fraction of the branches contains a fault. Hence, branch coverage under 100% does not provide absolute confidence of program correctness. Table 4 shows the results after execution of test suite with size 200.

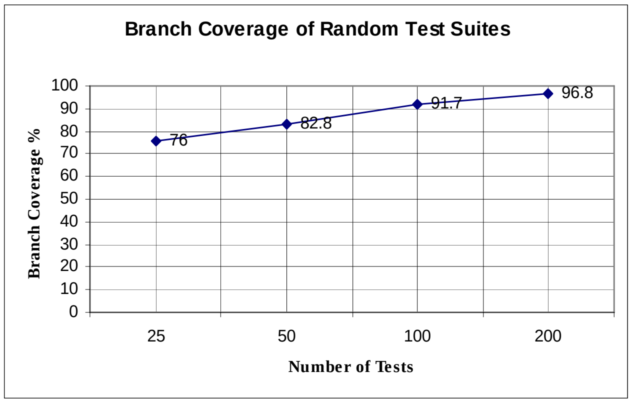


Figure 11. Branch coverage of randomly generated test suites

Table 4. Results of randomly generated test suite

Number Of Tests	Branch Coverage	Line Coverage	Execution Time
200	96.8%	99.2%	25.8 sec

As one can see, the execution time of the test suite is about six times greater that the one needed for running the tests obtained by symbolic execution. Note that presented results are average of fifty runs and vary slightly from run to run. Moreover, the time needed for generation of the symbolic and random test suites are less than a second. In conclusion, using random strategy imposes larger test suite, and which is more important, does not guarantee full coverage.

There is one more possible approach, though unrealistic, for testing the GUI—exhaustive testing. It is enough for one to consider the possible input space for the program to see that this is not an option. For example, the Workout Generator has inputs state of 2 x 3 x 3 x 100 x 1000 x 1000 = 1.8 x 10^9 combinations. The large input space is a result of the large number of possible values for the text box controls (and they only take numeric input). For example it is possible for one to use increments of 10 for the age and a range from 0 to 100 years, increments of 20 for the weight and range from 0 to 200 kilos, increments of 20 for the height and range from 0 to 200 centimeters. This would significantly reduce the input space but does not guarantee full coverage and still the possible inputs are 2 x 3 x 3 x 10 x 10 x 10 = 1.8 x 10^4. To explore this option a test suite using the upper conditions is generated and run on the GUI. Results are presented in Table 5.

Table 5. Results of randomly generated test suite

Number Of Tests	Branch Coverage	Execution Time
18000	88.8%	4671.88 sec

It is enough for one to compare the execution time to the one needed to run the tests obtained by symbolic execution or random testing to be convinced of the inapplicability of this approach. Moreover, because of the chosen increment factor some branches were left unexecuted.

Our results show a significant advantage of the test suite generated by symbolic execution over random generation in terms of branch coverage and size. However, one should notice that this technique is effective for GUIs that take user input and their execution follows different paths according to this input. Even though many GUIs do not belong to this class, a methodology of testing such programs is needed and no testing technique proposes a mechanism of dealing with such GUIs.

Another advantage of our technique is that during symbolic execution, for all unreachable paths, there are generated corresponding text files describing the constraints upon inputs that are unsatisfiable, in this way revealing code errors and facilitating their localization and correction. This is the opposite of testing with random values where such faults could remain unnoticed, and even though one has less than full code coverage, it is possible that all reachable code is executed. In such a case increasing the number of tests would not lead to any improvement. These constant results might be a clue to unreachable code, but the program has to be inspected manually in order to reveal such faults.

6. RELATED WORK

To the best of our knowledge the technique of symbolic execution is not applied in GUI testing. Because of this lack of related work, the following section is limited to investigating several approaches for GUI testing and some areas where symbolic execution is used for test input generation, program verification and testing.

In his Ph.D. Thesis [8] 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, assessment on GUIs. The author represents GUIs as a graph and obtains test cases performing different traversals on that graph. One weakness of this approach is that there in no technique for the generation of user stream input. Input values are read from a predefined data-base. Descriptions of the main components of that framework with further optimizations and improvements of the process may be found in [9], [10], [12], [13].

Memon, Banarjee and Nagarajan present a framework for regression testing of nightly/daily builds of GUI applications [11]. 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. Since this tool is based on the one in [8] it suffers the same limitation concerning user input streams.

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]. However, this approach has the same limitations as the previous ones.

A technique that transforms GUIs into a FSM and uses different techniques to reduce the states of the FSM and avoid state space explosion is proposed in [16]. However, here again the authors are focusing only on collaborating selections and user sequences over different objects in the GUI. Again the problem of the user input is not taken into consideration.

Forrester and Miller conduct an empirical study for reliability of GUI programs for Windows NT [4]. Authors test thirty different GUIs on Windows NT by using random streams of keyboard input, mouse events, and Win32 messages and observe if the application crashes or hangs. This technique shows the presence of bugs in about fifty percent of the tested applications but no information about what fragment of the code was problematic is provided.

Symbolic execution for test data generation is used in [14]. The program is represented as a deterministic FSM and using symbolic execution test data is generated. This paper deals exclusively with numeric constraint and examined programs are not GUIs.

The technique of symbolic execution is also used for program verification. It is applied for verification of safety-critical systems [3]. The paper proposes a framework that could be used for verification of code written in safe-C, which is the language used for most safety-critical systems.

Khurshid, Pasareanu and Visser use traditional symbolic execution and translate a program source to source thus allowing symbolic execution to be performed by a model checker [5]. This permits testing programs manipulating complex data structures to avoid the problem of state space explosion.

Symbolic execution is also performed on standard library classes [6]. The authors use an abstract representation of these classes as symbolic objects and define the semantics for operations on them. This way they avoid unnecessary symbolic execution of the code of standard library classes.

7. FUTURE WORK

As seen from the previous sections the proposed approach leads to increasing the effectiveness of GUI test suites. However, to perform the symbolic execution, the tester needs to instrument the code under test manually. We plan to explore ways of automating the process for C#, similar to what is presented in [7] where Java byte code is instrumented. The ultimate goal is a framework that performs the following steps: analyzing the GUI for applicability of symbolic execution, automated instrumentation of the GUI code, execution of the instrumented version, test generation and execution. Further, the implementation of symbolic strings needs to be improved and the constraint solver used by Barad needs to be upgraded. Another issue that should be addressed is a refinement of test case reduction. Even though the current implementation reduces successfully and significantly the number of tests, there is still room for improvement and make Barad more effective.

8. CONCLUSIONS

The main contribution is introducing the use of symbolic execution for GUI testing. Even though some aspects of the process of symbolic execution implemented in Barad need optimization, it was capable of handling all the constraints in the tested GUIs and our results show that the idea of using symbolic execution in GUI testing provides significantly better performance compared to random input generation in terms of line and branch coverage. It is also capable of capturing modifications that are to be made on the GUI in order to execute a particular segment of code. However, this approach does not pretend to replace traditional ones. Rather, it complements them by providing a technique for testing a class of GUIs that they are not capable to effectively verify. We believe that combining our approach with frameworks such as [8] and [11] would be beneficial and is worth exploring. Our technique addresses their main weakness and we deem that this would facilitate the development of a complete framework capable to handle all types of GUIs. Even though our results are encouraging, it would be useful to apply our approach to more applications in order to rigorously demonstrate its effectiveness.

9. REFERENCES

[1] Andrews, T., Quadeer, S., Rajamani, K., Rehof,.J., and Xie, Y. Zing:A model checker for Concurrent Software. In *Computer Aided Verification (ISBN: 978-3-540-22342-9)*, Springer Berlin / Heidelberg, Berlin, 484-487, 2004.

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

[3] Coen-Porisini, A., Denaro, G., Ghezzi, C., and Pezzè, M. Using symbolic execution for verifying safety-critical systems. In *ESEC / SIGSOFT FSE 2001*, 142-151, 2001.

[4] Forrester, J.E., and Miller, B.P. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *4th USENIX Windows Systems Symposium*, Seattle, August 2000.

[5] Khurshid, S., Pasareanu, C., and Visser, W. Generalized Symbolic Execution for Model Checking and Testing. In *9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003)*, Warsaw, Poland. Apr 2003.

[6] Khurshid, S., and Suen, S. Generalizing Symbolic Execution to Library Classes. In *6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, Lisbon, Portugal. Sep 2005.

[7] Khurshid, S.,Garcia, I., and Suen, I. Repairing Structurally Complex Data. *12th International SPIN Workshop on Model Checking of Software (SPIN)*, San Francisco, CA. Aug 2005.

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

[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), 2003, 260-269.

[10] 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.

[11] 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).

[12] 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).

[13] Memon, A. Using Tasks to Automate Regression Testing of GUIs. *In IASTED International Conference on ARTIFICIAL INTELLIGENCE AND APPLICATIONS (AIA 2004),* Innsbruck, Austria, Feb. 16-18, 2004. (BibTeX).

[14] 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

[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] 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.

[17] "Section 1: history of writing technologies" http://imrl.usu.edu/OSLO/technology_writing/004_003.htm Date Accessed: 02 December 2006.

[18] "NunitForms windows.forms unit testing" http://nunitforms.sourceforge.net/ Date accessed: 02 December 2006.

[19] "TestDriven.net" http://www.testdriven.net/overview.aspx> Date Accessed: 02 December 2006.

[20] "What is Java PathFinder?" http://javapathfinder.sourceforge.net/ Date accessed: 02 December 2006.

[21] "CVC3 Home Page" http://www.cs.nyu.edu/acsys/cvc3/> Date accessed: 02 December 2000.