GUI Testing as a Service

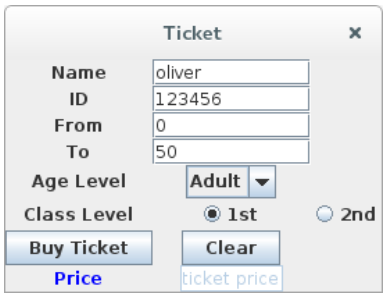

Figure 1: GUI for train ticket purchase

ABSTRACT

abstract

1. INTRODUCTION

Graphic User Interface(GUI) applications are driven by user events.

Manually testing of GUI is labor intensive.

`Guitar` [1] uses an automatic method to test GUI. It automatically generate event sequences for java GUI applications. But it use built-in concrete value for user inputs. Although it also accepts alternative user inputs, however itself doesn't provide any facility to generate user inputs.

`Barad` [2] is a novel GUI testing framework based on symbolic execution. It can obtain data inputs through executing symbolically test cases which are chains of event listener method invocations. It manually created symbolic mirror of java GUI library, so its released source code contains a large symbolic java GUI library. however it lacks necessary documentation, it is very hard to deploy the project, and such a heavy code base might be hard to maintain or error pruning.

`jfp-awt` [3] is an extension of the Java PathFinder for java GUI application model checking. It needs the user to specify sets of interaction sequences, for a real GUI application, this might be labor intensive task for testers.

In the paper we present a novel tool `Guicat` which generate concrete user inputs to cover different program branch. We build our work on top of `Guitar` and `Catg` [4].

2. MOTIVATING EXAMPLE

The GUI shown in Figure 2 is an application that we developed to mimic the motivating example used in Barad []. calculate the price for a train ticket. The application calculates the price for a train ticket based on the information

```
maxPrice =
minPrice =
int coefficient = (classLevel == TicketModel.
   FIRSTCLASS) ? 2 : 1;
int dist = to - from;
if (ageLevel == 1) {
  if (dist < 40) {
    price = 100 * coeficient;
  } else if (dist < 45) {
    price = 110 * coeficient;
  } else if (dist < 50) {
    price = 120 * coeficient;
  } else if (dist < 70) {
    price = 140 * coeficient;
  } else if (dist < 80) {
    price = 150 * coeficient;
  } else if (dist < 85) {
    price = 155 * coeficient;
  } else if (dist < 100) {
    price = 160 * coeficient;
  }
}
assert(price<=maxPrice&&price>=minPrice);
```

Figure 2: Code snippet for calculating ticket price

passed through the GUI. A user must provide a passenger class, name, ID, age level, and begin and end points. Each passenger class has its own coefficient that is used during the calculation. Each age level has a different base price depending on the distance to be traveled, which is the difference between From mile and To mile. Figure The calculation logic has twenty two branches with conditional statements nested three levels deep. The execution of a particular branch depends on the user input both in the form of data and event sequence (i.e. selecting a radio button).

This program is used to calculate ticket price according to different properties of the client, such as the class level, or age, or the travel distance. When the `Buy` button is clicked, the application check the `Name` and `ID` input, if the length of the `Name` string is less or equal than 3, or `ID` equals to some special string, then the application will display an error message and will not calculate any price. After checking the user information successfully, the application start to calculate the price as follows: read the user class input and set a coefficient for the price, read the user age to go to different pricing branch, read the start and dest input and calculate the distance of travelling, then calculate the price. We can see from the following code snippets that there should be many branches in the code to calculate ticket price for all kinds of clients.

One can not claim this kind of applications are well tested until all the branches has been covered during test stage.

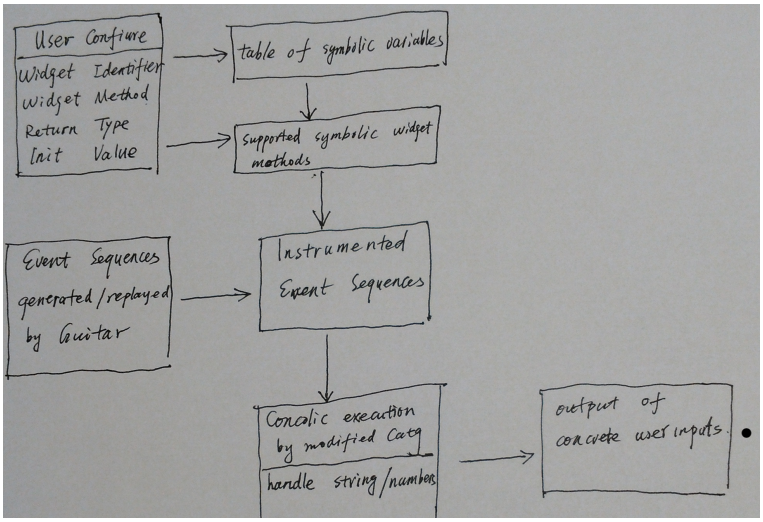

Figure 3: architecture

3. ALGORITHMS

Algorithm 1 TestGeneration(program GUI, int B, int S)

```
1: SET chains = N_0.ProduceEventSequence(B);
2: for all c ∈ chains do
3:    N_i = AllocateTask();
4:    tests.add(N_i.SymbolicExecution(c));
5: end for
```

4. IMPLEMENTATION

4.1 Architecture

The architecture is described as Figure 3

Our goal is to generate user inputs for java GUI applications to achieve high branch and code coverage.

The **work-flow** is we let the tester to create a configuration file, indicating which variables should be symbolic. Based on the configure file we create symbolic variables and symbolic functions which are used to replace the corresponding one in the **replayed sequences**. We generate replayed sequenses from guitar. Using guitar replayer to replay one event sequence is a replayed sequence. We do the replacement by `javaagent` using `ASM` the same way as `CATG` does. After replacement, we get an instrumented replayed sequence, then we can sent it to catg to do concolic execution. we use shell scripts to glue up modules for the testing process of an AUT.

We handle the problem that for a number input, java GUI framework returns a String to programmer by built in function *getText()*, and the programmer somewhere parse the String to numbers using *Integer.parseInt* or something similar. Because `CATG` can't handle this kind of operation, we modify catg to support casting functions like *Integer.parseInt*.

Note that catg requires the tester to set up symbolic variables in source code, and this may be inconvenient when source code is not available. We have automated this process.

The following are details of different modules

- **configuration file** contains entries of symbolic variable information. Each entry contains four elements separated by comma: widget identifier, widget method, return type, init value. A widget identifier is the *accessibleName* of the java widget, it is set by *setAccessibleName* or *setLabelFor*. Jpf-awt uses the same identifier. (Not certain whether guitar uses the same one or not). A widget method is the method which should be replaced by a symbolic one. It is usually a built in function which is used to get user input like *getText, isSelected, getActionCommand*. A return type is the type of this symbolic variable. A feasible return type should be one value in this list: String,int, long, char, byte, short, boolean. A init value is the concrete value of the symbolic variable for the symbolic execution to start with.

- **table of symbolic variables** is a set of symbolic variables created using the configuration file. It is stored in a *LinkedListedMap* to keep the order of the variables. For each entry in the configuration file, we create exactly one entry in the map whose key is the widget identifier, value is an symbolic entry object. If the return type is String, the symbolic entry object contains a symbolic string variable, if the return type is something else like an int, the symbolic entry object contains a symbolic string variable and a symbolic int variable. To the tester, the symbolic int variable is hidden, the output will write back to the String variable.

- **supported symbolic functions** is a collection of manually created functions which are used to replace the corresponding ones in the replayed sequences. For example, we create a static function *SGetText(Object)* for the *getText()* of *javax.swing.TextInput*. A little trick here is to pass the object to the static function parameter so that the stack frames don't need change.

- **instrumentation** We use ASM to instrument java bytecode. We replace the function call with a symbolic one if it satisfies the following conditions: 1. the call is from an object which has an widget identifier exists in the configuration file; 2. the function name exists in the configuration file

- **parseInt support**. We add parseInt support to catg. When the symbolic string is transformed to int by *parseInt*, catg doesn't support this function originally, so falls to concrete execution. We modify catg to add this support. When the catg encounters *parseInt*, we check the string. if the string is one in the symbolic table, and there is an int symbolic variable in the same symbolic entry, we use this int symbolic value as the return value of catg's symbolic call *parseInt*, so catg can continue execute symbolically. We have modified 3 files in catg's originmal source.

5. EXPERIMENT

We have tested ticket seller. We user guitar to generate testcase. Each guitar test case we have get up to 13 concrete inputs.

if we set guitar event sequences length to 1, then guitar will genenrate 10 testcases, we can generate 82 testcases. if we set guitar event sequences length to 2, then guitar will genenrate 89 testcases, we can generate 952 testcases.

References

[1] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. Guitar: an innovative tool for automated testing of gui-driven software. *Automated Software Engineering*, 2014.

[2] Svetoslav Ganov, Chip Killmar, Sarfraz Khurshid, and Dewayne E. Perry. Event listener analysis and symbolic execution for testing gui applications. *Formal Methods and Software Engineering*, 2009.

[3] Peter Mehlitz, Oksana Tkachuk, and Mateusz Ujma. Jpf-awt: Model checking gui applications. *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, 2011.

[4] Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. Tesma and catg: automated test generation tools for models of enterprise applications. *IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.