Symbolic Execution for GUI Testing

Svetoslav Ganov, Sarfraz Khurshid, Dewayne Perry
{svetoslavganov, khurshid, perry}@ece.utexas.edu

Challenges in GUI testing:

- ➢ Selection of event sequences
- ➢ Selection of values for widgets

Classic approaches:

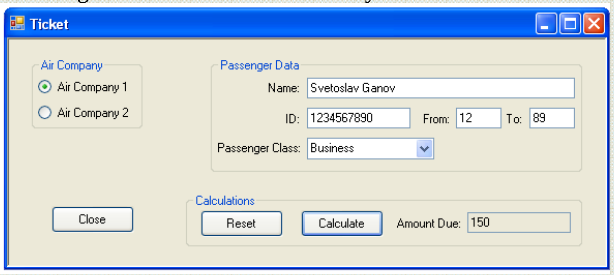
- ➢ Focus on event sequences abstracting the GUI as Finite State Machine or a graph and generate test cases traversing these structures
- ➢ Either do not consider data dependant behavior or use manually selected values

Our approach:

- ➢ Is a white-box testing approach
- ➢ Symbolically executes the GUI code and generates a test suite that maximizes code coverage while minimizing the number of tests needed to systematically check the GUI
- ➢ Addresses data-flow as well as event-flow of a GUI application

Example:

- ➢ The application:
- - Calculates amount due for a plane ticket
- - Behavior depends on the user input
- - Program execution tree has twenty three branches



Air Company: (● Air Company 1) ○ Air Company 2
Name: Svetoslav Ganov
ID: 1234567890 From: 12 To: 89
Passenger Class: Business
Amount Due: 150

- ➢ Results:

Table1. Results of symbolically generated test suite

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

Table2. Results of randomly generated test suite

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

```
void ReduceTests(Collection<Test> tests, Test test,
                                        bool canAppend)
{
  foreach(Test t in tests){
    if(t.HasValues()&&!t.Equals(test)&&!t.IsTerminal)){
      if(t.Vars().Intersect(test.Vars()).IsEmpty()
                                          && canAppend)
          t.Append(test);
      }
      else
        t.TryMerge(test);
    }
  }
}
```

Figure 1. Test reduction algorithm

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.

Framework overviews:

1) Instrumentation of the GUI application using symbolic classes provided by Barad's libraries
2) Execution of the instrumented code
3) As result from the symbolic execution a set of log files and a test suite are generated
4) Execution of the test suite and generate a report

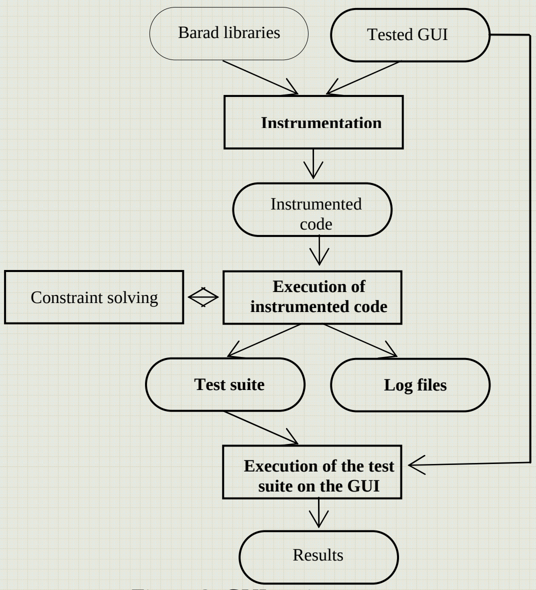


Figure 2. GUI testing process

Conclusions:

- ➢ Our prototype Barad provides significantly better performance compared to previous approaches in terms of line and branch coverage.
- ➢ Our technique handles GUI applications that the previous approaches are not capable to effectively verify.
- ➢ Combining our technique with existing frameworks presents a very promising approach for systematic testing of GUIs.