A Case for GUI Testing Using Symbolic Execution
Poster Abstract

Svetoslav Ganov, Sarfraz Khurshid, Dewayne Perry
University of Texas at Austin
{ganov, khurshid, perry}@ece.utexas.edu

A Graphical User Interface (GUI) 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.

Key Challenges in GUI Testing.

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 inputs. To illustrate, consider testing a GUI with one textbox that takes a ten character string as input. Exhaustive testing requires 26^{10} possible input strings (assuming we limit each character to be from the English alphabet in lower-case).

Traditional GUI Testing

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 [2]. 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.

Symbolic Execution for GUI Testing

We envision a novel GUI testing approach based on symbolic execution [1]. Our goal is to generate values for data widgets and address data-flow as well as event-flow of a GUI application. By symbolically executing the underlying GUI code a test suite that maximizes code coverage while minimizing the number of tests needed to systematically check the GUI could be obtained. During symbolic execution all reachable paths of the program are explored and (for decidable constraints) infeasible paths are detected. For each feasible path an event sequence could be generated as a test.

To illustrate the potential benefit of using symbolic execution, we used random generation to obtain values for the text input widgets of a small GUI application that we developed. The widgets take String as well as primitive integer values. We provide tight bounds on the sets of possible values to assist random testing perform optimally. Our results show that for achieving 95% branch coverage a test suite with two hundred tests is needed. A quick inspection of the source code shows that the GUI code has only fifty-four branches. Since symbolic execution visits at least one new branch during each path it explores, using symbolic execution to generate tests requires at most one test to cover each branch, significantly reducing the test suite size and the time to test. Additionally, symbolic execution can help with minimizing previously generated tests, for example by removing tests that are subsumed by other tests in the suite.

We are currently implementing Barad, our GUI testing framework based on symbolic execution.

References

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

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