Test Case Generation

To generate a set of test cases, test designers attempt to cover all the functionality of the system and fully exercise the GUI itself. The difficulty in accomplishing this task is twofold: to deal with domain size and with sequences. In addition, the tester faces more difficulty when they have to do regression testing.

Unlike a CLI (command line interface) system, a GUI has many operations that need to be tested. A relatively small program such as Microsoft WordPad has 325 possible GUI operations.

In a large program, the number of operations can easily be an order of magnitude larger.

The second problem is the sequencing problem. Some functionality of the system may only be accomplished with a sequence of GUI events. For example, to open a file a user may have to first click on the File Menu, then select the Open operation, use a dialog box to specify the file name, and focus the application on the newly opened window. Increasing the number of possible operations increases the sequencing problem exponentially. This can become a serious issue when the tester is creating test cases manually.

Regression testing becomes a problem with GUIs as well. A GUI may change significantly, even though the underlying application does not. A test designed to follow a certain path through the GUI may then fail since a button, menu item, or dialog may have changed location or appearance.

These issues have driven the GUI testing problem domain towards automation. Many different techniques have been proposed to automatically generate test suites that are complete and that simulate user behavior.

Most of the testing techniques attempt to build on those previously used to test CLI (Command Line Interface) programs, but these can have scaling problems when applied to GUI's.

For example, Finite State Machine-based modeling — where a system is modeled as a finite state machine and a program is used to generate test cases that exercise all states — can work well on a system that has a limited number of states but may become overly complex and unwieldy for a GUI

Planning and artificial intelligence

A novel approach to test suite generation, adapted from a CLI technique involves using a planning system. Planning is a well-studied technique from the artificial intelligence (AI) domain that attempts to solve problems that involve four parameters:

- an initial state,
- a goal state,
- a set of operators
- a set of objects to operate on

Planning systems determine a path from the initial state to the goal state by using the operators. As a simple example of a planning problem, given two words and a single operation that replaces a single letter in a word with another, the goal might be to change one word into another.

Authors used the planner IPP to demonstrate this technique. The system's UI is first analyzed to determine the possible operations. These become the operators used in the planning problem. Next, an initial system state is determined, and a goal state is specified that the tester feels would allow exercising of the system. The planning system determines a path from the initial state to the goal state, which becomes the plan of test.

Using a planner to generate the test cases has some specific advantages over manual generation. A planning system, by its very nature, generates solutions to planning problems in a way that is very beneficial to the tester:

The plans are always valid. The output of the system is either a valid and correct plan that uses the operators to attain the goal state or no plan at all. This is beneficial because much time can be wasted when manually creating a test suite due to invalid test cases that the tester thought would work but didn’t.

A planning system pays attention to order. Often to test a certain function, the test case must be complex and follow a path through the GUI where the operations are performed in a specific order. When done manually, this can lead to errors and also can be quite difficult and time consuming to do.

Finally, and most importantly, a planning system is goal oriented. The tester is focusing test suite generation on what is most important, testing the functionality of the system.

When manually creating a test suite, the tester is more focused on how to test a function (i. e. the specific path through the GUI). By using a planning system, the path is taken care of and the tester can focus on what function to test. An additional benefit of this is that a planning system is not restricted in any way when generating the path and may often find a path that was never anticipated by the tester. This problem is a very important one to combat.

Another method of generating GUI test cases simulates a novice user. An expert user of a system tends to follow a direct and predictable path through a GUI, whereas a novice user would follow a more random path. A novice user is then likely to explore more possible states of the GUI than an expert.

The difficulty lies in generating test suites that simulate ‘novice’ system usage. Using Genetic algorithms have been proposed to solve this problem.

Novice paths through the system are not random paths.

First, a novice user will learn over time and generally won’t make the same mistakes repeatedly, and, secondly, a novice user is following a plan and probably has some domain or system knowledge.

Genetic algorithms work as follows: a set of ‘genes’ are created randomly and then are subjected to some task. The genes that complete the task best are kept and the ones that do not are discarded. The process is again repeated with the surviving genes being replicated and the rest of the set filled in with genes that are more random. Eventually one gene (or a small set of genes if there is some threshold set) will be the only gene in the set and is naturally the best fit for the given problem.

In the case of GUI testing, the method works as follows. Each gene is essentially a list of random integer values of some fixed length. Each of these genes represents a path through the GUI. For example, for a given tree of widgets, the first value in the gene (each value is called an allele) would select the widget to operate on, the following alleles would then fill in input to the widget depending on the number of possible inputs to the widget (for example a pull down list box would have one input… the selected list value). The success of the genes are scored by a criterion that rewards the best ‘novice’ behavior.

A system to do this testing for the X window system, but extensible to any windowing system is described in.

The X Window system provides functionality (via XServer and the editors' protocol) to dynamically send GUI input to and get GUI output from the program without directly using the GUI. For example, one can call XSendEvent to simulate a click on a pull-down menu, and so forth. This system allows researchers to automate the gene creation and testing so for any given application under test, a set of novice user test cases can be created.

Running the test cases

At first, the strategies were migrated and adapted from the CLI testing strategies.

Mouse position capture

A popular method used in the CLI environment is capture/playback. Capture playback is a system where the system screen is “captured” as a bitmapped graphic at various times during system testing. This capturing allowed the tester to “play back” the testing process and compare the screens at the output phase of the test with expected screens. This validation could be automated since the screens would be identical if the case passed and different if the case failed.

Using capture/playback worked quite well in the CLI world but there are significant problems when one tries to implement it on a GUI-based system.[8] The most obvious problem one finds is that the screen in a GUI system may look different while the state of the underlying system is the same, making automated validation extremely difficult. This is because a GUI allows graphical objects to vary in appearance and placement on the screen. Fonts may be different, window colors or sizes may vary but the system output is basically the same. This would be obvious to a user, but not obvious to an automated validation system.

Event capture

To combat this and other problems, testers have gone ‘under the hood’ and collected GUI interaction data from the underlying windowing system.[9] By capturing the window ‘events’ into logs the interactions with the system are now in a format that is decoupled from the appearance of the GUI. Now, only the event streams are captured. There is some filtering of the event streams necessary since the streams of events are usually very detailed and most events aren’t directly relevant to the problem. This approach can be made easier by using an MVC architecture for example and making the view (i. e. the GUI here) as simple as possible while the model and the controller hold all the logic. Another approach is to use the software's built-in assistive technology, to use an HTML interface or a three-tier architecture that makes it also possible to better separate the user interface from the rest of the application.

Another way to run tests on a GUI is to build a driver into the GUI so that commands or events can be sent to the software from another program.

This method of directly sending events to and receiving events from a system is highly desirable when testing, since the input and output testing can be fully automated and user error is eliminated.