Test Case Generator for GUITAR

Daniel R. Hackner
Department of Computer Science
University of Maryland, College Park, MD 20742
E-mail: dhackner@umd.edu

Atif M. Memon
Department of Computer Science
University of Maryland, College Park, MD 20742
E-mail: atif@cs.umd.edu

ABSTRACT
As Graphical User Interfaces become more popular, the need for GUI testing tools becomes greater. However, many current test generation techniques require proprietary tools and can be hard to use to their fullest potential. This paper outlines a new test case generation strategy, which enables testers to automatically produce cases in a widely used format. It is the hope of the authors that this strategy will encourage more complete GUI testing throughout the field.

Keywords
Graphical user interfaces, jfcUnit, integration tree, JUnit, event-flow, test case generation, GUI ripping, code coverage

1. INTRODUCTION
The GUITAR (GUI Testing frAmewoRk) project is supported by the National Science Foundation with the purpose of simplifying GUI testing by automatically creating test cases that intelligently challenge a GUI's functionality. The objective of our research was to allow for generation of test cases in a widely used format, without limiting GUITAR's test coverage. This has been accomplished by translating GUITAR's proprietary format test cases into jfcUnit test cases.

2. GUI TESTING FIELD
2.1 Context
Software testing is a crucial element of software development, necessary to encourage software usability, robustness, and safety. Effective testing is very time consuming and can account for upwards of 67% of the total cost of software development.[1] While the field of software testing is very comprehensive, the subfield of GUI testing represents a relatively new specialization. The variations of GUI design and functionality are innumerable, making GUIs hard to predict and challenging to test.[2]

2.2 Necessity
While facially simple to the user, the underlying complexity of GUIs make them susceptible to errors. GUIs complicate software engineering because they add an additional interface, which in itself can contain errors, onto the underlying code. GUITAR aims to reduce the complexity of GUI testing, specifically in situations where the source code might not be available. As the use of GUIs in software programming is rapidly increasing, the necessity for versatile GUI testing tools increases as well.

2.3 Difficulty
GUI testing is uniquely challenging in that the tester may be limited to the abstraction layer that the GUI represents graphically, rather than the source code that lies beneath it. Finding GUI objects and their relationships to each other in an interface, without access to the source code is difficult.

As an illustration of GUI testing complexity, Microsoft WordPad has 325 GUI operations.[3] Even though there are very few components, a test may yield different results depending upon the order and number of prior component activations.

3. GUITAR PROJECT OVERVIEW
3.1 GUI Ripper
The most valuable component of the GUITAR process is the GUI Ripper. The reverse engineering of an executing GUI is called "ripping" a GUI. This tool examines a GUI hierarchically, creating an "Integration Tree" of the GUI elements that shows their relationship to each other and to the GUI as a whole. "Starting from the software's first window (or set of windows), the GUI is 'traversed' by opening all child windows. All the window's *widgets* (building blocks of the GUI, e.g., buttons, text-boxes), their *properties* (e.g., background-color, font), and *values* (e.g., red, Times New Roman, 18pt) are extracted."[4] Once the GUI structure is understood, objects can be manipulated and tested accordingly.

3.2 EFG Generator
Another component, the EFG Generator, collects data from a GUI to create an event-flow graph. Event-flow graphs seek to demonstrate all of the possible interactions between GUI object events at any given time.[5] When done manually, determining the relationship between objects can be the most time intensive part of testing. However, GUITAR eliminates this highly time-consuming task. "Once the event-flow model is created, it can be used to generate a large number of GUI test cases with very little cost and effort."[6]

3.3 Test Case Generator
GUITAR's Test Case Generator uses all of the information in an Integration Tree and an Event-Flow graph to create a set of proprietary tests that manipulate the GUI. Two of the generation methods by which it does this are node selection and edge selection. In node selection, the Test Case Generator picks a random event from the event-flow graph and then ascertains the component manipulation steps that are required to reach the event. Edge selection works in a similar fashion by running node selection on one of an edge's nodes, and then adding the other node to the path. The cases are then outputted, including information such as the component's name, and the way that the component should be manipulated. Previously, such test cases had to be run through a proprietary re-player which made it difficult to automate efficiently. The focus of our current research is to translate these proprietary test cases into a more widely accessible format.

```
<Component>
<Window>2Pad - Untitled *_0</Window>
<Nonterminal>Justification_1</Nonterminal>
<Eventtype>LEFTCLICK</Eventtype>
<Eventvalue>UNKNOWN</Eventvalue>
</Component>
```

Figure 3. Portion of a random GUITAR test case[7]

3.4 Coverage Evaluator

GUITAR is able to analyze the percentage of executed lines in application code by generating statement coverage, branch coverage and path coverage reports.

4. jfcUnit

jfcUnit is an extension to JUnit which is a software framework used in the creation of automated tests for programs written in Java. By providing specialized GUI component finders that locate Swing objects on the screen, as well as component activators that access a Swing component's various functions (mouse click, keyboard input, etc.), jfcUnit adds Java Swing (Java GUI) testing functionality to JUnit. Since JUnit is powerful, well known and widely used, jfcUnit has a very broad appeal by association and therefore, is an optimal way to output GUI test cases.

5. jfcUnit TEST CASE GENERATOR

5.1 Scope

The scope of the Test Case Generator was to convert the output of GUITAR GUI rips and test cases into robust test sets in a widely used format. Since GUITAR's test cases were originally outputted in a proprietary and lesser known format, testers were compelled to utilize GUITAR's own tools. This constraint limited GUITAR's appeal and usefulness. The need for a more popular format is now met by the ability to convert output into jfcUnit cases. jfcUnit was chosen because of it's ease of use and portability. With the Generator, a tester can input a GUI into GUITAR and obtain output in the form of jfcUnit cases.

5.2 Caveats

As of this writing, jfcUnit can only find objects that are visible on screen. Thus, the Generator requires traversing through the GUI rips in order to find methods of activating the appropriate buttons to make the components appear. This can occasionally lead to cases which are difficult to read, consisting of numerous mouse click calls on objects, or circular pop-up activations, before the desired object is found. The major limitation of the Generator is that upon starting the GUI, all pop-ups are killed, which can become an issue if the component in question was located on that window. We are searching for a method to allow original pop-ups to remain opened if needed. As of this writing, components can only be manipulated and tested if they lie on the main screen or the screens reachable from within it.

5.3 Details and Methodology

Each test case is analyzed by a Python script, which finds its components and the actions to be applied to them. Since components can be nested (e.g. the Save button under the File menu) and invisible upon the starting of a GUI, the GUI rip files must be examined in order to find the relationship of objects to each other. These paths can vary for the same component, as GUITAR finds numerous routes to access the same element. This algorithm provides a deeper level of bug searching, as it delivers unique event combinations. To implement this in the jfcUnit test cases, the script determines the series of steps required to access a specific component from the start of its window. If the window is not the main screen, the program searches for elements that can invoke that window. One of these elements is recursively run again, in order to find the steps necessary to activate it. The process is repeated until a path to the main screen is found. The script then outputs each test case, along with a GUI activator, which will analyze the GUI itself in order to find the way to initially start it. Finally, these cases are compiled into a folder.

```
1: JMenuItem jmiComponent;
2: JMenuItemFinder jmiFinder = new JMenuItemFinder("Edit",
        true);
3: jmiComponent = (JMenuItem) jmiFinder.find();
4: assertNotNull( "Could not find Edit!", jmiComponent);
5: getHelper().enterClickAndLeave(new MouseEventData(this,
        jmiComponent));

6: jmiFinder.setText("Split Word");
7: jmiComponent = (JMenuItem) jmiFinder.find();
8: assertNotNull( "Could not find Split Word!", jmiComponent);
9: getHelper().enterClickAndLeave( new MouseEventData(this,
        jmiComponent));
```

Figure 5. Example generated jfcUnit test

This is a portion of a sample test case outputted by the Generator. Each four line set finds a component and activates it. The “Split Word” set is dependent on the “Edit” set executing properly.

5.4 Contributions

jfcUnit cases are written in Java, which is renowned for its system portability. The Test Case Generator provides a tester with easy to generate system test cases, as well as skeleton code which can be used in the writing of more specialized tests. Since jfcUnit cases can be verbose and difficult, having basic, environmentally independent cases already made and ready for customization is valuable. This Generator is able to seek out objects within nested menus and pop-ups, which is extremely powerful for complex GUIs. The test case layout is divided into subsections, each of which manipulates a specific object. This format makes each step's location abundantly clear, and thus permits alterations and expansions to be easily made.

5.5 Present Status

As the project is now completed, GUITAR can create jfcUnit test cases with the ability to perform in-depth component existence checks, with support for nested menus and pop-ups, as well as to manipulate objects by clicking or entering keyboard input. There is still a lack of support for components located on original pop-ups. We hope to have this issue resolved in time for presentation.

6. ACKNOWLEDGEMENTS

Thanks to Xun Yuan and the team for help and guidance.

7. REFERENCES

[1] http://www.cs.umd.edu/~atif/GUITARWeb/

[2] Memon, A. M. 2007. An event-flow model of GUI-based applications for testing: Research Articles. *Softw. Test. Verif. Reliab.* 17, 3 (Sep. 2007), 137-157. DOI= http://dx.doi.org/10.1002/stvr.v17:3

[3] Memon, A. M., Pollack, M. E., and Soffa, M. L. 1999. Using a goal-driven approach to generate test cases for GUIs. In *Proceedings of the 21st international Conference on Software Engineering* (Los Angeles, California, United States, May 16 - 22, 1999). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 257-266.

[4] Memon, A., Banerjee, I., and Nagarajan, A. 2003. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In *Proceedings of the 10th Working Conference on Reverse Engineering* (November 13 - 17, 2003). WCRE. IEEE Computer Society, Washington, DC, 260.

[5] Memon, op. cit. 2

[6] Ibid

[7] http://www.cs.umd.edu/~atif/GUITARWeb/guitar_process_test_case_generation.htm

DEMONSTRATION

The demonstration will consist of an end-to-end run through of GUITAR, along with a slideshow presentation to highlight the project's main points. An example GUI will be selected, and used as input. A set of jfcUnit cases will be outputted, and these examples will be displayed. Afterwards, the examples will be run, to show cases of test failure and success upon multiple bug scenarios. The demonstrated cases can test the multitude of options available, such as nested menus and different component inputs. Cases will be chosen specifically to show the variety of cases that GUITAR can create. Questions will then be fielded from the audience.

In addition to showing the test results, one test case will be selected and demonstrated step-by-step. The following example tests the JMenuItem entitled "Preferences"[1] and shows how a sample generated jfcUnit case checks it by first opening and closing the "New Crossword" menu:

```
public void testPreferences_R_6() throws Exception
{
    JMenuItem JMenuItemComponent;

    JMenuItemFinder jmiFinder = new JMenuItemFinder("File", true);
    JMenuItemComponent = (JMenuItem) jmiFinder.find();
    assertNotNull( "Could not find File!", JMenuItemComponent);
    getHelper().enterClickAndLeave( new MouseEventData( this, JMenuItemComponent ));
```

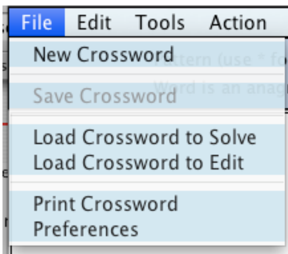

Figure 1. "File" menu is opened

```
    jmiFinder.setText("New Crossword");
    JMenuItemComponent = (JMenuItem) jmiFinder.find();
    assertNotNull( "Could not find New Crossword!", JMenuItemComponent);
    getHelper().enterClickAndLeave( new MouseEventData( this, JMenuItemComponent ));
```

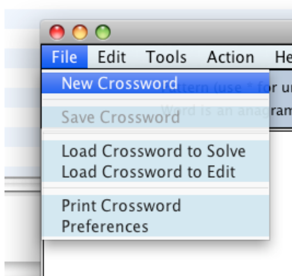

Figure 2. "New Crossword" is selected

```
    JButton JButtonComponent;

    AbstractButtonFinder jbuttonFinder = new AbstractButtonFinder("Cancel", true);
    JButtonComponent = (JButton) jbuttonFinder.find();
    assertNotNull( "Could not find Cancel!", JButtonComponent);
    getHelper().enterClickAndLeave( new MouseEventData( this, JButtonComponent ));
```

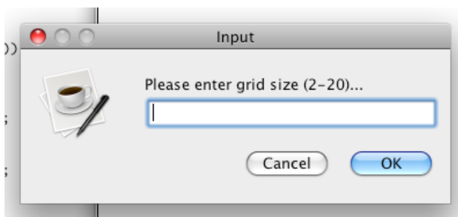

Figure 3. Input box is closed

```
    jbuttonFinder.setText("OK");
```

```
JButtonComponent = (JButton) jbuttonFinder.find();
assertNotNull( "Could not find OK!", JButtonComponent);
getHelper().enterClickAndLeave( new MouseEventData( this, JButtonComponent ));
```

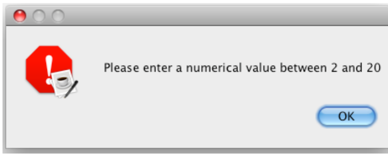

Figure 4. Error message is closed

```
jmiFinder.setText("File");
JMenuItemComponent = (JMenuItem) jmiFinder.find();
assertNotNull( "Could not find File!", JMenuItemComponent);
getHelper().enterClickAndLeave( new MouseEventData( this, JMenuItemComponent ));
```

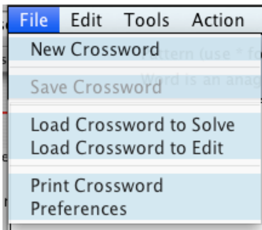

Figure 5. "File" menu is selected again

```
jmiFinder.setText("Preferences");
JMenuItemComponent = (JMenuItem) jmiFinder.find();
assertNotNull( "Could not find Preferences!", JMenuItemComponent);
getHelper().enterClickAndLeave( new MouseEventData( this, JMenuItemComponent ));
```

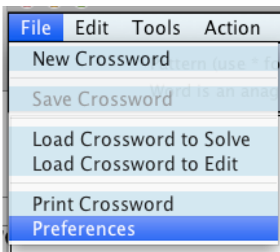

Figure 6. Preferences option is selected

```
}
```

This particular test case works quite well, as it found a bug that occurred from closing the "New Crossword" menu.

[1] http://sourceforge.net/projects/crosswordsage