GUITAR AMP

(**A**uto**M**atic jfcunit test case **P**roducer)
http://www.cs.umd.edu/~atif/ugprojects/Francis_and_Hackner/GUITAR_AMP.html

Daniel R. Hackner -
dan.hackner@gmail.com

Advisor: Atif M. Memon -
atif@cs.umd.edu

Abstract

This paper describes a new method of generating test cases for a Java GUI. Many current methods require cases to be written by hand, or the use of tools that can be slow and tedious in setup. Our goal was to create a versatile system that can be used with relatively little setup and configuration.

Introduction

GUITAR is a longstanding project funded by the National Science Foundation with the goal of alleviating problems generally associated with GUI testing. The GUITAR team believes that by making GUI testing tools simple to use, we can greatly improve the quality and process of developing GUIs. The objective of AMP was to seamlessly move from a proprietary output format into a much more general and accepted form.

It is hoped that this transition will encourage more people to use the GUITAR toolset, by easing the developer's transition from current practices to the GUITAR method.

GUITAR Project

Previous GUITAR Work

Originally, GUITAR consisted of four major components: the GUI Ripper, EFG Generator, Test Case Generator and the Coverage Evaluator. AMP has been added to complement the original tools.

The GUI Ripper may be the most important element of the GUITAR system; it reverse engineers the layout of a running GUI in order to ascertain its format. This component creates an "Integration Tree" of the GUI components, which shows the interrelations of the components and how they nest within each other. The process consists of repeatedly traversing through the child windows of the GUI root. A GUI must be ripped so that the tester can visualize and test the components.

The EFG (event-flow graph) Generator creates a graph from the GUI, which shows all possible sequences of GUI events that can occur. It is within this stage that the tools begin to calculate the domain of all component execution sequences. "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."[1]

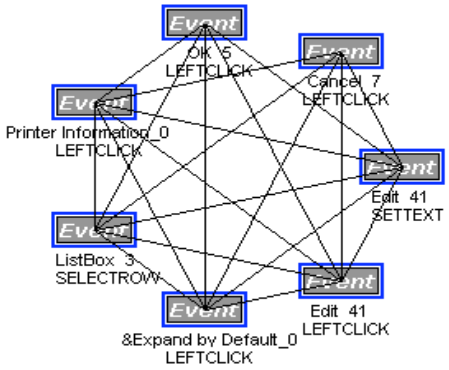

Figure 1. Event-Flow Graph for *WordPad-->ConnectToPrinter*[2]

The Test Case Generator combines the Integration Tree and the Event-Flow Graph to create tests that manipulate the GUI. These tests are in a proprietary format that can only be interpreted by GUITAR tools. Test case "paths" are produced via graphing techniques. The Generator picks random events or edges from the event-flow graph and searches for the component manipulations that are required to connect them. This execution path, along with information about the components and how to manipulate them are written into proprietary cases. The goal of this research was to translate these cases from the proprietary format into one that is more widely usable.

The Coverage Evaluator is a way for GUITAR to examine the coverage percentage and usefulness of the test code. It does this through statement coverage, branch coverage and path coverage reports.

GUITAR AMP Addition
As mentioned above, GUITAR AMP was created to alleviate the limited scope and compatibility issues of the generated test cases. We decided to create a conversion tool to transform the proprietary cases into jfcUnit cases. jfcUnit is a GUI testing extension of the widely used JUnit framework. This change will allow our test cases to be used on any platform with Java and a few readily available Java extensions. Once AMP is installed, a user can use the GUITAR suite to create a jfcUnit test suite from a GUI file.

AMP was written primarily in Python because of its superior text parsing tools. A small portion was written in Java to work in conjunction with jfcUnit. During execution, the script reads each test case and determines the actions to be performed on each component. Once all of the appropriate paths have been found, a test case is outputted with code that invokes the .jar file.

Frequently, necessary components are nested within others and AMP must examine the GUI rip files in order to add the necessary steps to uncover them. The most trouble lies within window components because AMP must locate a component that can invoke the window. If this located component is also not currently available, then AMP must continue to recurse up the GUI structure until an available component is found.

One of the most important improvements in version 2.0 of the software was the creation of the “UnnamedFinder.java”. Previously, the tools could only manipulate components that developers had named. This proved to be a problem with textboxes and other components that were frequently unnamed. Since many test cases require input to textboxes, the tests would fail because the components could not be found without names. The new unnamed component finder allows AMP to identify components based on class, window depth and container. Now that the unnamed component finder has been added to AMP, the outputted jfcUnit cases are significantly more complete, rendering fewer false positives for bugs.

Tutorial
Visit http://www.cs.umd.edu/~atif/GUITARWeb/ for instructions on how to use GUITAR.

Pre-Run Warning: These directions should all be run atomically on the same system. GUITAR generated files from Windows platforms are likely to not work with AMP on Unix platforms, as the system’s built-in GUI components are different.

1) Locate the .jar file for the software to be tested.
2) Run GUITAR on the .jar file and get a .gui rip file as well as a set of .tst files.
3) Pick the .tst file to be converted into jfcUnit .
4) Install Python 2.5 or above. Install JUnit and JFCUnit.
5) Run AMP using the command flags as noted in the software. As of version 2.0 use the following:
Command line args (Strings must be in quotes):
"--testSuite": Mandatory. A full path to the location of the .tst test suite file
"--gui": Mandatory. A full path to the .gui rip file
"--jarPath": Mandatory. A full path to the .jar file
"--package": Optional. If provided, all created files will be made in the provided package
"--kill": Optional. Use this to signal that there are initial popups which appear upon opening this application (such as a version notification).
"--debug": Optional. Gives debugging output.
"--multiPath": Optional. Searches for all possible paths to a GUI component. Significantly slower to generate the test cases, but can result in slightly more optimal paths.

Example run flag input on a Unix system:
--kill=1
--package="crosswordsage"
--jarPath="/Users/dhackner/Documents/workspace/CrosswordSage.jar"
--testSuite="/Users/dhackner/Documents/workspace/100.tst"
--gui="/Users/dhackner/Documents/workspace/crosswordsageGUI.gui"

This will output into the *.jarFileName*jfcUnitTests folder.

6) Check that the Java Build Path of the project which contains the generated jfc files has JUnit, JFCUnit and the path of the .jar file in test.
7) Execute the .java file as a JUnit test.

Accomplishments

Version 2.0 of the software has recently been completed, and will be integrated with the full GUITAR project in the near future. A research paper on GUITAR AMP was presented at the 2008 International Conference on Software Engineering in Leipzig, Germany. There was a great deal of interest shown in the tool, as well as in the future of the entire project.

Acknowledgements
I would like to express thanks to Xun Yuan and Jaymie Strecker for their help, as well as the rest of the GUITAR team for providing me with the tools to base this work on.

References
[1] “An event-flow model of GUI-based applications for testing” by Atif M. Memon. Software Testing, Verification and Reliability, vol. 17, no. 3, 2007, pp. 137-157, John Wiley and Sons Ltd.. http://www.cs.umd.edu/~atif/papers/MemonSTVR2007.pdf

[2] http://www.cs.umd.edu/~atif/GUITARWeb/guitar_process_efg.htm

Example
The following is an example test for the open source application “crosswordsage” (http://crosswordsage.sourceforge.net/), generated by version 2.0 of the tool. It works alongside UnnamedFinder.java

```
import java.util.jar.*;
import javax.swing.*;
import junit.extensions.jfcunit.*;
import junit.extensions.jfcunit.finder.*;
import junit.extensions.jfcunit.eventdata.*;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class CrosswordSagejfcUnitTest_50 extends JFCTestCase
{
   private JFCTestHelper helper = null;

   public CrosswordSagejfcUnitTest_50(String name)
   {
      super(name);
   }

   private void callMainMethodOfJarFile(String jarName) throws
ClassNotFoundException, NoSuchMethodException, IllegalAccessException,
InvocationTargetException, FileNotFoundException, IOException
```

```
        JarInputStream jarIn = new JarInputStream(new FileInputStream(jarName));
        Manifest manifest = jarIn.getManifest();
        if (manifest == null) {System.err.print(jarName + " does not have a manifest file.");}
        Attributes a = manifest.getMainAttributes();
        String mainClass = a.getValue("Main-Class");
        Class cl = Class.forName(mainClass);
        Class[] argTypes = {String[].class};
        Method main = cl.getDeclaredMethod("main", argTypes);
        main.invoke(null, ((Object)new String[0]));
    }

    protected void setUp() throws Exception
    {
        super.setUp( );
        setHelper( new JFCTestHelper( ) ); // Uses the AWT Event Queue.
        callMainMethodOfJarFile("/Users/dhackner/Documents/workspace/GUITAR
Research/CrosswordSage.jar");
        flushAWT();
        java.util.List list = new DialogFinder(null).findAll();
        for(int i = 0; i < list.size(); i++) //close all windows beyond the first one
        {
            getHelper().disposeWindow((JDialog)list.get(i), this);
        }
    }

    protected void tearDown() throws Exception
    {
        super.tearDown( );
        flushAWT();
    }

    public void test50() throws Exception
    {
        JMenuItemFinder jmenuitemFinder = new JMenuItemFinder("", true);
        JMenuItem jmenuitemComponent;

        UnnamedFinder unnamedFinder = new UnnamedFinder("", true);
        java.awt.Component component;

        AbstractButtonFinder jbuttonFinder = new AbstractButtonFinder("", true);
        JButton jbuttonComponent;
```

```
        jmenuitemFinder.setText("File");
        jmenuitemComponent = (JMenuItem) jmenuitemFinder.find();
        assertNotNull( "Could not find a JMenu called File!", jmenuitemComponent);
        getHelper().enterClickAndLeave(new MouseEventData(this,
jmenuitemComponent));

        jmenuitemFinder.setText("New Crossword");
        jmenuitemComponent = (JMenuItem) jmenuitemFinder.find();
        assertNotNull( "Could not find a JMenuItem called New Crossword!",
jmenuitemComponent);
        getHelper().enterClickAndLeave(new MouseEventData(this,
jmenuitemComponent));

        unnamedFinder.setStats(0, "BasicOptionPaneUI$MultiplexingTextField", "Input");
        component = (java.awt.Component) unnamedFinder.find();
        assertNotNull( "Could not find a BasicOptionPaneUI$MultiplexingTextField called
AutoText!", component);
        getHelper().sendString( new StringEventData( this, component, "5") );

        jbuttonFinder.setText("OK");
        jbuttonComponent = (JButton) jbuttonFinder.find();
        assertNotNull( "Could not find a JButton called OK!", jbuttonComponent);
        getHelper().enterClickAndLeave(new MouseEventData(this, jbuttonComponent));

        jmenuitemFinder.setText("Edit");
        jmenuitemComponent = (JMenuItem) jmenuitemFinder.find();
        assertNotNull( "Could not find a JMenu called Edit!", jmenuitemComponent);
        getHelper().enterClickAndLeave(new MouseEventData(this,
jmenuitemComponent));

        jmenuitemFinder.setText("Split Word");
        jmenuitemComponent = (JMenuItem) jmenuitemFinder.find();
        assertNotNull( "Could not find a JMenuItem called Split Word!",
jmenuitemComponent);
        getHelper().enterClickAndLeave(new MouseEventData(this,
jmenuitemComponent));

        unnamedFinder.setStats(0, "BasicOptionPaneUI$MultiplexingTextField", "Input");
        component = (java.awt.Component) unnamedFinder.find();
        assertNotNull( "Could not find a BasicOptionPaneUI$MultiplexingTextField called
AutoText!", component);
        getHelper().sendString( new StringEventData( this, component, "5") );
```

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

        jmenuitemFinder.setText("File");
        jmenuitemComponent = (JMenuItem) jmenuitemFinder.find();
        assertNotNull( "Could not find a JMenu called File!", jmenuitemComponent);
        getHelper().enterClickAndLeave(new MouseEventData(this,
jmenuitemComponent));

        jmenuitemFinder.setText("Save Crossword");
        jmenuitemComponent = (JMenuItem) jmenuitemFinder.find();
        assertNotNull( "Could not find a JMenuItem called Save Crossword!",
jmenuitemComponent);
        getHelper().enterClickAndLeave(new MouseEventData(this,
jmenuitemComponent));

        unnamedFinder.setStats(12, "MetalFileChooserUI$3", "Save");
        component = (java.awt.Component) unnamedFinder.find();
        assertNotNull( "Could not find a MetalFileChooserUI$3 called AutoText!",
component);
        getHelper().sendString( new StringEventData( this, component, "5") );

    }//end of function

}//end of testCase
```