# An Experimenter's Guide to the GUITAR Infrastructure

Jaymie Strecker and Penelope Brooks

October 16, 2008

## 1 Introduction

GUITAR is the GUI Testing frAmewoRk. Currently, documentation for GUITAR exists in this manual and in the Software-Testing Benchmarks at http://www.cs.umd.edu/ atif/Benchmarks/. This manual can be obtained from the Software-Testing Benchmarks page. An extended version is available to members of the GUITAR group in CVS module `GUITAR/docs/manual`.

## 2 Ripping a GUI

### 2.1 Building JavaGUIRipper

TODO

### 2.2 Running JavaGUIRipper

To generate a GUI model for the application under test, do:

```
java -cp JavaClassFiles[:APPCLASSPATH] JavaGUIRipper -s
```

for standalone mode. Better yet (see the section on workarounds), do:

```
java -cp JavaClassFiles[:APPCLASSPATH] JavaGUIRipper -g GUIFILE -c MAINCLASS [-a
MAINCLASSARGS] [-u URL] [-w INITIALWAIT] [-e EXCEPTIONFILE] [-i IGNOREFILE]
```

The classpath of the application under test can be specified either in `APPCLASSPATH` (preferred; see the section on workarounds) or as a semicolon-separated, quoted list of file URLs (each starting with "file:") in `URL`. When stripped of the URL formatting, `URL` ends up the same as `APPCLASSPATH`. `GUIFILE` is the GUI file where the output will go. `MAINCLASS` is the name of the main class of the application under test, and `MAINCLASSARGS` is a semicolon-separated, quoted(?) list of arguments (`ARG1;ARG2;...`) to pass to the main class. JavaGUIRipper essentially invokes the application under test like this:

```
java -cp APPCLASSPATH MAINCLASS ARG1 ARG2 ...
```

The last three arguments to JavaGUIRipper are not passed along to the application under test. `INITIALWAIT` is the number of milliseconds to wait (*e.g.*, for a splash screen to disappear) before starting to rip. `EXCEPTIONFILE` and `IGNOREFILE` each list exceptions to be made to JavaGUIRipper's rule that a GUI component is clicked on during ripping if and only if its label ends in "..." (*e.g.*, a menu item called "Open..."). By convention, the "..." suffix indicates that the GUI component causes a new window to open, but not all applications follow this convention. `EXCEPTIONFILE` lists the label of each *additional* GUI component that should be clicked on during ripping even though its label does not end in "...". `IGNOREFILE` lists the label of each GUI

component that ends in "..." but, for whatever reason, should *not* be clicked on during ripping. In both files, the labels should be separated by linebreaks.

You may want to modify the GUI file for various reasons—for example, to remove windows that are redundant or events that you do not want to execute during testing (such as Print). To remove a window, just delete its entry in the GUI file and modify the Invokelist property of any widgets that invoke it. To remove an event, change the "_R_" in its name to "_N_" and delete its ReplayableAction property in the GUI file.

### 2.3 Errors and workarounds

In JavaGUIRipper's standalone mode, the "Spy" button is unreliable and only available under Windows. The workaround is to use the non-standalone mode and specify an `EXCEPTIONFILE`.

If a `java.lang.ClassNotFoundException` occurs, and the class not found belongs to the application under test, then the application's classpath may not have been specified completely in `APPCLASSPATH` or `URL`. Passing the classpath directly to Java (via `APPCLASSPATH`), rather than indirectly (via `URL`), is more reliable.

## 3 Generating an EFG or EIG

To generate an EFG from a GUI file, do:

```
java [-Xmx512m] -cp JavaClassFiles:JavaClassFiles/jdom.jar:JavaClassFiles/xercesImpl.jar
EFG -g GUIFILE [-e EFGFILE]
```

For an EIG, instead do:

```
java [-Xmx512m] -cp JavaClassFiles:JavaClassFiles/jdom.jar:JavaClassFiles/xercesImpl.jar
EIG -g GUIFILE [-e EIGFILE]
```

The `Xmx512m` option allocates more memory than normal to the JRE. `GUIFILE` is the GUI file to be read in, and `EFGFILE` (`EIGFILE`) is the EFG (EIG) file where the output will go. If this option is not specified, the output is sent to a file in the same directory as `GUIFILE` (e.g. if `GUIFILE` is `/tmp/ex.GUI` the output goes to `/tmp/ex.EFG`).

## 4 Errors and workarounds

The error message "TERMINAL widget has no calling widget" occurs when a Widget element with value "TERMINAL" for property "Type" is not reached from (i.e., its enclosing Window element is not a value of) the "Invokelist" property of any other Widget element. Often this occurs because the Widget element's "Type" property value should actually be something else, such as "SYSTEM INTERACTION". In this case, edit the GUI file accordingly.

In CVS, scripts `Jaymie_ExptOSS/scripts/summarize_gui`, `Jaymie_ExptOSS/scripts/summarize_efg`, and `Jaymie_ExptOSS/scripts/unsummarize_efg` can help diagnose and fix errors at this stage. Also in CVS, the program `Jaymie_BigExpt/scripts/CheckEventReachability.java` (which must be run with java's "-ea" option) can identify unreachable events in the event graph.

# 5 Generating test cases

## 5.1 Building the test-case generators

TODO

## 5.2 Running the test-case generators

```
java [-Xmx512m] -cp JavaClassFiles (StructuralTestCaseGenerator|CompleteTestCaseGenerator)
-g GUIFILE [-e (EFGFILE|EIGFILE)] -n NUMBER -l LENGTH (-r|-s) [-d TARGETDIR]
```

`StructuralTestCaseGenerator` generates test cases whose events all belong to the same window. Test cases generated by `CompleteTestCaseGenerator` can exercise more than one window per test case. In both generators, the `-r`/`-s` option controls the order in which test cases are generated (randomly or sequentially, respectively).

The `Xmx512m` option allocates more memory than normal to the JRE. `GUIFILE` and `EFGFILE` (`EIGFILE`) are the locations of the GUI and EFG (EIG) files used to generate test cases. If no EFG or EIG file is given, one is generated on the fly. If `NUMBER` is positive, `NUMBER` test cases of length `LENGTH` are generated; if `NUMBER` is 0, all possible test cases of length `LENGTH` are generated. The generated test cases are written to files in `TARGETDIR`, which defaults to the directory from which the test-case generator was invoked.

## 5.3 Errors and workarounds

`StructuralTestCaseGenerator` ignores the `NUMBER` given; it always generates all test cases. To generate fewer test cases, interrupt `StructuralTestCaseGenerator` before it is finished (*e.g.*, with Ctrl-C). This problem does not occur with `CompleteTestCaseGenerator`.

# 6 Instrumenting code to get coverage data

## 6.1 Statement coverage

This section describes JGuitarInstrumentor, a component of GUITAR that instruments Java classes to track statement coverage. Other tools to track statement coverage (and additional coverage metrics) that have been used successfully with GUITAR include Emma (http://emma.sourceforge.net) and JCoverage (http://www.jcoverage.com/).

### 6.1.1 Building the instrumentor

From CVS, check out `GUITAR/src/JGuitarInstrumentor` and `GUITAR/src/Util`. Use the included makefiles (in the `code` directory of each project) to compile them (`Util` first) to `GUITAR/bin/JavaClassFiles/`.

### 6.1.2 Instrumenting an application

Since `JGuitarInstrumentor` expects the `JavaClassFiles` directory to be on the top level of your working directory, you must `cd` to `Guitar/bin/`. Do

```
java -cp JavaClassFiles JGuitarInstrumentor
```

This displays a GUI (Figure 1), which lets you pick the Java source files you want to instrument.

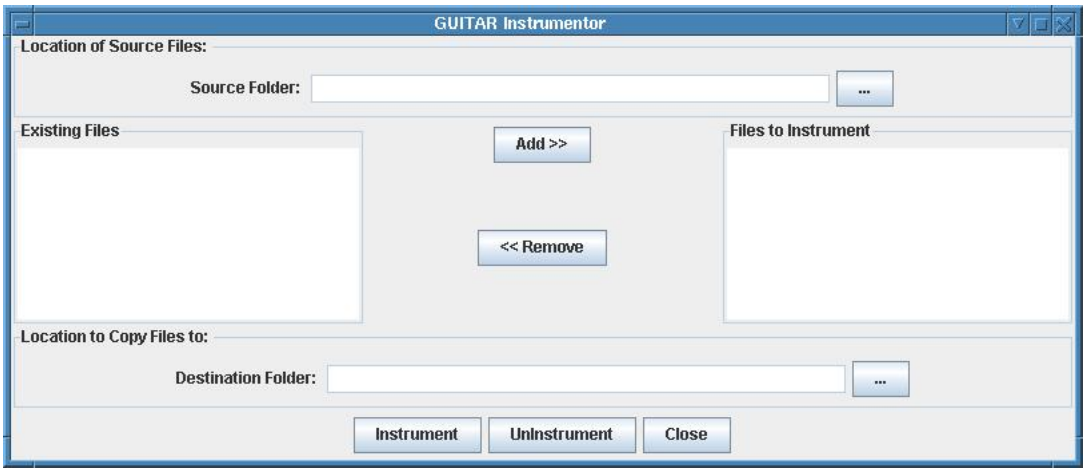


Figure 1: JGuitarInstrumentor GUI

On Unix, an error is reported in the shell:

```
/bin/sh:  instr2.cmd:  execute permission denied
```

You need to do

```
chmod u+x instr2.cmd
```

to make it executable. If there are any resources (e.g. JAR files) that need to be in the classpath for compiling the application, add them to `instr2.cmd`. Now do

```
./instr2.cmd
```

The `JGuitarInstrumentor` class is a wrapper around an instrumentor called `instr`. Documentation for `instr` may be found at http://www.glenmccl.com/instr/instr.htm. Although `instr` can track both statement and method coverage, `JGuitarInstrumentor` only looks at statement coverage.

An alternative to using `JGuitarInstrumentor` is to instrument and compile the the application on the command line. First, copy the source files and resources (e.g. JAR files) for the application into a new directory. Now run the instrumentor on the copied source files, following the instructions at http://www.glenmccl.com/instr/instr.h Finally, compile the instrumented source files, using the application's makefile if it has one.

If you move an application to a different directory after instrumenting it, be aware that the `_prof.prof` class may refer to the application's files with absolute paths. If this is the case, the simplest remedy is to copy the uninstrumented application to the desired new application and instrument it there.

### 6.1.3 Collecting coverage data

By default, when you run the instrumented application, the coverage information isn't printed. To have it printed, add the call `_prof.prof.write()` to the instrumented source code. This isn't necessary when running the application through JavaGUIReplayer, since the replayer calls `_prof.prof.write()` itself.

JavaGUIReplayer looks for the `_prof` directory in its own parent directory, not in the application's directory. Hence, `_prof` must be copied from the application's directory to the `JavaClassFiles` directory.

### 6.1.4 Errors and Workarounds

When instrumenting large applications, `_prof/prof.java` may turn out to be too large to compile. The workaround is modify `_prof/prof.java` so that entries in the array `lns` (and possibly `files`) are read from a separate file.

If `_prof.prof.write()` may be called more than once per test case (e.g., after each step of the test case), then, since the call to `instr.ProfData` alters `files`, `lns`, and `cnt`, the values of these variables before the call should be restored after the call, and the method should be made synchronized. For example:

```
public static synchronized void write(String fileName)
{
    String[] filesCopy = new String[files.length];
    int[] lnsCopy = new int[lns.length];
    System.arraycopy(files, 0, filesCopy, 0, files.length);
    System.arraycopy(lns, 0, lnsCopy, 0, lns.length);
    instr.ProfData p = new instr.ProfData(files, lns, cnt, true);
    p.writeOut(fileName, false);
    files = filesCopy;
    lns = lnsCopy;
    java.util.Arrays.fill(cnt, 0);
}
```

## 6.2 Dataflow coverage

InsectJ, an instrumentation framework, can be used to instrument Java classes to track dataflow coverage (in addition to other types of coverage such as branch, method, and class).

### 6.2.1 Installing the instrumentor

Follow the instructions at http://insectj.sourceforge.net/ to download and install the InsectJ Eclipse plug-in.

### 6.2.2 Instrumenting an application

The documentation at http://insectj.sourceforge.net/ provides the technical details of the instrumentation process. Briefly, the steps are as follows:

1. In a new Eclipse project, write and build one or more monitors (Java classes) that implement the `MonitorObject` and `DefMonitorInterface` and/or `UseMonitorInterface` interfaces. In each monitor's `processData` method, which is called when the instrumented program exits, print the coverage results to standard output or append them to a file. (Multiple monitor instances will be created when the program runs—hence the importance of *appending* to a file rather than overwriting.) See the InsectJ documentation for details. Be aware that `getName()` and `getCanonicalName()` return null for anonymous classes.

2. Close the monitors project and restart Eclipse. Because of a bug in InsectJ, you must restart Eclipse each time you switch projects to reset the list of available monitors shown in the `Run...` dialog in step 7.

3. Create an Eclipse project for the program you want to instrument. To be compatible with InsectJ, the project must use the Java 5 compiler. Make sure the project builds and runs without the instrumentation.

4. Add InsectJ to the project's classpath by right-clicking on the project icon and selecting the `Add InsectJ to classpath` option.

5. Import the monitors project (as a package of class files or as a JAR) to the current project.

6. Create a directory called `instrumented` in the top-level directory of the current project, and copy all of the (uninstrumented) class files into `instrumented`. This is the easiest way to ensure that classes you choose not to instrument, as well as interfaces—which can't be instrumented—end up alongside the instrumented class files.

7. Open the `Run...` dialog and click on the `Instrumentation` tab. Add the `DefProbeInserter` and `UseProbeInserter` to the project. Select the checkboxes for all classes and methods you want to instrument. The checkboxes can act unpredictably sometimes, so carefully make sure that everything you intended to check is indeed checked. (Click on the triangles to fully expand the tree view of the project classes and methods.) Select the options to save the configuration to a file (`insectconfig.xml`) and the instrumented classes to a directory (`instrumented`).

8. Click `Run`. Note that InsectJ instruments classes on-the-fly when they are loaded, so a class file's instrumented counterpart is only placed in the `instrumented` directory if the class is loaded during program execution. You may need to interact with the program a bit to cause each class to be loaded; check timestamps in `instrumented` to ensure that each class file you intend to instrument is indeed instrumented.

InsectJ is distributed with several sample monitors, including a `SimpleDefUseMonitor` class. These monitors may suffice, depending on the task, or at least they may serve as templates for home-grown monitors.

#### 6.2.3 Collecting coverage data

To run the instrumented program outside of Eclipse, do

```
java -cp PROBES/probes.jar -jar RUNTIME/runtime.jar CONFIGFILE PROJECTDIR
```

`PROBES` and `RUNTIME` are the parent directories of InsectJ's `probes.jar` and `runtime.jar` files. `CONFIGFILE` is the location of the InsectJ configuration file (e.g. `insectconfig.xml`). `PROJECTDIR` is the directory that contains the instrumented class files (e.g. `instrumented`).

# 7 Replaying test cases

## 7.1 Building JavaGUIReplayer

## 7.2 Replaying on one machine

TODO JavaGUIReplayer, replay_scripts

### 7.3 Replaying on the cluster

TODO

### 7.4 Determining test case outcomes

TODO Oracle(Info)Verifier, OracleInfoParser

### 7.5 JavaGUIReplayer design

TODO

### 7.6 JavaGUIReplayer errors and workarounds

TODO

### 7.7 OracleVerifier, OracleInfoVerifier, and OracleInfoParser errors and workarounds

TODO

## 8 JavaProfiler

### 8.1 Building JavaProfiler

From CVS, check out `GUITAR/src/User-Profiles`. Use the included makefile (in the project's `code` directory) to compile the code to `GUITAR/bin/JavaClassFiles`.

### 8.2 Running JavaProfiler

```
java -cp User-Profiles[:APPCLASSPATH] Guitar DELAY
```

`APPCLASSPATH` is the classpath of the application being profiled. `DELAY` is the delay (in milliseconds) used in `ControllerforProfiler.run()`. Currently, the main class of the application to be profiled is hard-coded into `Guitar.loadclass()`, but it is not difficult to change it or make it a parameter of JavaProfiler.

### 8.3 JavaProfiler design

Although JavaProfiler is written in Java, the design is primarily procedural. There are just three public classes:

- `theStackforProfile`. With no apparent benefit, this class re-implements `java.util.Stack`.
- `Guitar`. The `main` method of this class calls `Guitar.loadclass()` and `Guitar.startRecording()` in sequence. These methods invoke (via reflection) the main class of the application being profiled and kick off the capture-and-record process. Unfortunately, the main class of the profiled application is hard-coded to be the main class of GUITAR, but it is easy to make this an argument to JavaProfiler. *Probably duplicates parts of GUITAR.*

- `ControllerforProfile`. The interesting stuff happens here.

    1. `run()`. With length-`delay` pauses in between, the profiler traverses the GUI components of the active window. A `delay` value of 500 (milliseconds) works well. Calls `ActiveWindow()` and `GetComponentFromWin()`.
    2. `ActiveWindow()`. Finds the currently-active GUI window. *Probably duplicates parts of GUITAR.*
    3. `GetComponentFromWin()`. If the active window hasn't already been instrumented with event handlers, instrument it (via reflection). Calls `getComponentVectorforWindow()` to grab the list of things to instrument. Adds the event handlers using `addDynamicMenuListeners()` and `addDynamicCompListeners()`.
    4. `getComponentVectorforWindow()`. This traverses (depth-first) the components in the active window. Delegates some of this to `TraverseContainer()`. *Probably duplicates parts of GUITAR.*
    5. `Button_Action_Handler()`, `Text_Action_Handler()`, and `Menu_Action_Handler()`. These are the available event handlers, triggered by `myActionListenerforButton.actionPerformed()`, `myFocusListener.focusLost()`, and `myActionListenerforMenu.actionPerformed()`, respectively. Each event handler gets the list of components in the active window (`ActiveWindow()` and `getComponentVectorforWindow`), finds an integer index of the component based on its depth-first-search ord er (`getButtonLoc()`, `getTextLoc()`, and `getMenuLoc()`), and records the event in the log file (`LogToFile()`).
    6. `getButtonLoc()`, `getTextLoc()`, and `getMenuLoc()`. *Probably duplicates parts of GUITAR.*
    7. `LogToFile()`. *Probably duplicates parts of GUITAR.*

# 9 Mutation testing

## 9.1 Generating mutants

One way to generate mutants is with MuJava. To obtain MuJava, go to http://cs.gmu.edu/ offutt/mujava/ and follow the instructions to download and install. To run MuJava, change to the directory where MuJava is installed (the directory containing `mujava.config`, `mujava.jar`, and so on) and do:

```
java -cp classes:$CLASSPATH mujava.gui.GenMutantsMain
```

$CLASSPATH should include the paths specified in the MuJava installation instructions. The source files to mutate should be in a sub-directory of the MuJava directory called `src`, and the corresponding class files should be in `classes`, also a sub-directory of the MuJava directory.

## 9.2 Replaying mutants

To run, or replay a test case on, a mutated version of an application, either insert the path for the mutant class into the classpath before the regular path for the application or substitute the mutant class itself into the regular application. In the first case (altering the classpath), the mutant class will need to be placed in a directory structure corresponding to its package structure.

Along with directories `class_mutants` and `traditional_mutants`, MuJava creates for each class the directory `original`. This contains source and class files equivalent to, *but not identical to*, the original files in `src` and `classes`. For example, the formatting is different (pretty-printed), and some extra information is included (full package names for classes). The mutants are derived from this version of the class.

### 9.3 Errors and Workarounds

For some reason, MuJava refuses to operate on any class that extends a class in the `javax.swing` package. To get around this, one can construct a package `fakeswing` (in `classes`) and make the classes to mutate (in `src` and `classes`) extend `fakeswing` classes instead. Once mutants are generated, the original references to `javax.swing` need to be restored in the mutated source code, and the source code needs to be recompiled. See `fakeswing` in `/fs/guitar/MuJava_mutants/`, `recompile_mutants` in CVS module `Jaymie_ExptOSS/scripts`, and `cp_mujava_orig_files` in CVS module `Jaymie_BigExpt/scripts/`.

## 10 For more information...

TODO (GUITAR Web site, various publications)