gtklogger: A Tool For Systematically Testing Graphical User Interfaces

Stephen A. Langer and Yannick Congo*

Information Technology Laboratory

National Institute of Standards and Technology

Gaithersburg, MD, 20899

Andrew C.E. Reid, Rhonald C. Lua, and Valerie R. Coffman

Material Measurement Laboratory

National Institute of Standards and Technology

Gaithersburg, MD, 20899

R. Edwin García

School of Materials Engineering

Purdue University

West Lafayette, IN 47907-2044

(Dated: September 29, 2014)

Abstract

We describe a scheme for systematically testing the operation of a graphical user interface. The scheme provides a capability for generating event logs, which are recordings of a user session with the interface. These logs can be annotated with assertion statements, comparing reference test data with data retrieved by introspection on the GUI elements. Such an annotated log forms a test case, suitable for incorporation into a regression test suite.

1. INTRODUCTION

It is well established among software developers that systematic testing of software, whether against a formal design specification or against a less formal case-by-case assesment of correct functionality, is a critical part of the software development cycle, and crucial to the production of high-quality software.

Much formal testing methodology revolves around viewing software products as essentially functions, in the mathematical sense, abstractly taking a single input from some large domain, performing a computation, and producing a single output. Increases in both the raw power of computers and in the power and scope of software development tools have lead to increasing complexity in software. Presenting this complexity to the user in a comprehensible way demands a graphical user interface, which has its own complexity, and makes the software a stateful, interactive machine. While it is still true that the software can be thought of as a function, now the inputs and outputs have to be considered to be the full state of the program and its GUI. This complicates the application of traditional testing methods. At the same time, the GUI raises the bar for the expected reliability — it allows users more rapid access to potentially-overlooked corner cases, and encourages the expectation that all aspects of the program will "just work".

Contact with formal testing methodology can be restored by the introduction of systematic ways of testing of a graphical user interface. In this article, we take a step in that direction, describing the construction of a testing kit, that allows a user session with a GUI to be recorded and played back, and that allows the state of the GUI widgets to be queried at repeatable locations within the session, ensuring both correctness and consistency of the GUI state.

For our example, we use the GIMP Tool Kit (GTK) widget set[1], specifically gtk+, version 2.6 or later, and its Python wrappers, PyGTK[2]. We have created a Python module, called `gtklogger`, which can be used to record, replay, and test a program with a PyGTK user interface. `gtklogger` was developed specifically to test the OOF[3] project at NIST, but can easily be applied to other PyGTK programs. This paper includes instructions for extending `gtklogger` to handle PyGTK objects that were not used in OOF.

2. FUNCTIONAL OVERVIEW

The test kit we are constructing must be able to record and replay a user session by logging events to a file, and later reading that file and recreating the events. The first thing to decide is what kind of events should be logged. One possibility is to track every mouse motion, mouse click, and keyboard button press. However, our goal is not to test the components of a graphics library, but to test a program written to use that library. We assume that the graphics library itself is working correctly. The appropriate domain of the test kit is therefore the interface between the graphics library and the application. This approach has a number of advantages:

1. The number of events is greatly reduced. Moving the mouse into a window and clicking on a button is represented by a single 'click' event, rather than a series like ['enter window', 'move mouse', 'enter button', 'move mouse', 'click'].

2. The recorded events are independent of the position of the graphics windows and even of the position of interface components within the windows. The beneficial consequences of this are that (a) Redesigning the user interface will not necessarily invalidate existing test scripts, (b) Test scripts are portable, since they don't depend on any aspects of hardware, and (c) Test scripts are insensitive to other programs running simultaneously.

We are not the first to suggest this approach. In particular, Andersson and Bache[4] have proposed a similar scheme, also implemented for PyGTK. Our method operates at a slightly lower level of abstraction than this tool, generating actual Python code during the recording mode which is then run in replay mode. This allows us greater flexibility in manipulating and extending the replay functionality, at the price of a greater sensitivity to architectural changes in the GUI.

The operation of the system is shown schematically in figure 1, which shows the flow of control for both the recording and playback cases.

There are four steps involved in using the `gtklogger` module to test a PyGTK program. First, the program must be modified ("instrumented") so that it can record events. The `gtklogger` system provides a substitute for the standard "connect" functionality common

to many toolkits, including PyGTK. `gtklogger`'s substitute "connect" function is the hook through which events are recorded. This is described in detail in Sec. 2.1.

Second, the program must be run with recording enabled in order to generate a test case. In the simplest case, this is nearly identical to an ordinary run of the program, the only differences being the modifications to use the `gtklogger` "connect" scheme, and the setting of a global switch to turn on recording. The output of this process is a log file describing all of the recorded events which occurred during the run. Here, user or tester judgement is required to ensure that a useful test case is generated. This is described in greater detail in Sec. 2.2.

Third, the log file should have tests of the program state inserted into it, adding lines of Python code which compare the internal states of program objects, including PyGTK objects, with their correct state for the circumstances. This completes construction of the test case. The `gtklogger` scheme provides a number of tools for accessing PyGTK objects so that their internal state can be queried, and also additional tools for controlling the order of events in threaded applications. These tools and their use are described in Sec. 2.3.

Finally, the test is implemented by re-running the program, synthesizing the recorded events through the `gtklogger` replay mechanism, as described in Sec. 3.2. The test fails if any of the assertions in the modified log file raise errors, and passes if they do not. The robustness and suitability of the test is the responsibility of the test case generator. Each of the previous steps need be undertaken only once, but this final step can be treated as in standard regression testing, and re-run whenever the program is modified or ported to a different computer system.

2.1. Instrumenting a Program

A gtk+ application consists of a collection of `Widget`s, which are objects (in the object-oriented programming sense) that are drawn on the computer screen and with which the user can interact. Windows, buttons, labels, sliders, and text entry fields are all `Widget`s. Some `Widget`s are `Container`s, with other "child" `Widget`s within them. `Container`s can be nested, and every visible `Widget` can trace its ancestry through its parent containers back to a top-level widget, which is usually a `gtk.Window` or `gtk.Dialog`. `Widget`s interact with the program by emitting "signals", which are associated with callback functions via a `Widget`'s

`connect` method.

When using `gtklogger`, this basic structure doesn't change, but a few new function calls must be added and a few PyGTK function calls must be replaced by `gtklogger` variants. The modifications add a small amount of function-calling overhead associated with GUI events in comparison with the uninstrumented code, but in practice, do not affect the execution of the program in any way, except of course when log files are being recorded or replayed. There are a number of ways in which programs must be modified:

1. Widgets must be given identifying names.
2. Signals must be connected to callback functions via the function `gtklogger.connect` instead of the method `gobject.GObject.connect`.
3. Non-widget objects that emit signals must be *adopted* by named widgets.
4. Submenus must be added using the function `gtklogger.set_submenu` instead of the method `gtk.MenuItem.set_submenu`.
5. `gtk.Dialog`s must be replaced by `gtklogger.Dialog`s.
6. Checkpoints must be inserted.

These operations are described in more detail in the following subsections.

2.1.1. Naming Widgets

When `gtklogger` is replaying a line from a log file, it has to be able to identify the widget that caused the line to be recorded, in order to recreate the event. `gtklogger` identifies widgets by giving them names[5], and keeping track of all of the existing top-level widgets. Because container widgets know their children, knowing the name of a widget and all of its parents enables `gtklogger` to address the actual widget object. Names of top-level widgets are assigned like this:

```
window = gtk.Window()
gtklogger.newTopLevelWidget(window, 'windowname')
```

Names of other widgets are assigned with `setWidgetName`:

```
button = gtk.Button()
gtklogger.setWidgetName(button, 'buttonname')
```

Passive objects like `Frames` don't need to have names, but it can be convenient to give them names if it will make a log file more readable. Widgets are identified in log files by a colon-separated string of names, starting from a top level widget. For example, `'windowname:frame:button'` identifies a widget named `'button'` inside a widget named `'frame'` inside a top-level widget named `'windowname'`. There may be unnamed nested containers in the hierarchy between `'windowname'` and `'frame'` or between `'frame'` and `'button'`, but the string is a legal identifier for the button as long as there are no other widgets that could answer to the same name. For example, if a window contains a `gtk.VBox` that contains two `gtk.Frame`s that each contain one `gtk.Button`, the `Button`s can have the same name if the `Frames` have different names, or the `Frame`s can be unnamed if the `Button`s have different names.

2.1.2. Connecting to Signals

PyGTK `Widget`s interact with a program by emitting "signals", indicating that their state has been changed (either interactively by the user or programmatically via a function call). A program "connects" to an object's signals if it wants to be notified of changes in the state of the object. For example, the following code fragment puts a button in a window and calls a callback function, `buttonCB`, when the button is clicked:

```
window = gtk.Window()
button = gtk.Button('Press Here')
window.add(button)
button.connect('clicked', buttonCB)
```

`gtklogger` defines a new function, `gtklogger.connect`, which works like the gtk `connect` function, but also connects to a callback that logs events.[6] When using `gtklogger`, the above code snippet looks like this:

```
window = gtk.Window()                          # unchanged
gtklogger.newTopLevelWidget(window, 'Top')     # new line
```

```
button = gtk.Button('Press Here')              # unchanged
gtklogger.setWidgetName(button, 'red')         # new line
gtklogger.connect(button, 'clicked', buttonCB) # modified line
```

Each time the button is clicked, the line

```
findWidget('Top:red').clicked()
```

will be recorded in the log file, and `buttonCB` will be called. During playback, the line will be read, the `gtklogger` function `findWidget` will retrieve the named button, and the mouse click will be simulated by calling the button's `clicked` method.

Sometimes events that don't have callbacks need to be recorded and replayed. For example, if the appearance of the user interface depends on the window size, it will be necessary for `gtklogger` to record `'configure-event'` signals, even though the program might not otherwise catch that signal. `gtklogger` defines a `connect_passive` function to do this, like this:

```
window = gtk.Window(gtk.WINDOW_TOPLEVEL)
gtklogger.newTopLevelWidget(window, 'windowname')
gtklogger.connect_passive(window, 'configure-event')
```

`connect_passive` works just like `connect`, except that it doesn't take a callback function argument.

When `Widget`s have their state set by the program, rather than by a user, they generally should *not* call their callbacks, and should *definitely* not record their change of state in the `gtklogger` log file. This is because the log file records user actions. If a user action indirectly makes a widget change state, and if that change of state is recorded in the log file, then when the log is replayed the change of state will occur twice: once in response to the log entry for the original user action, and once for the log entry of the state change. Therefore it's necessary for logging to be suppressed before the program changes the state of a `GObject`. In normal operation, gtk+ signals are blocked and unblocked by calling an object's `handler_block` and `handler_unblock` methods, which take as an argument the return value of the corresponding `connect` call. Because `gtklogger.connect` actually makes two connections (one to the callback and one to the logging mechanism), `gtklogger` provides an easy way of blocking both connections at once. `gklogger.connect`

and gtklogger.connect_passive return a GUISignals object that has its own block and unblock methods. For example, the following snippet sets the state of a button without logging it:

```
button = gtk.CheckButton('On/Off')
gtklogger.setWidgetName(button, 'cuteasa')
buttonsignal = gtklogger.connect(button, 'clicked', button_callback)
...
buttonsignal.block()
button.set_active(1)
buttonsignal.unblock()
```

Cautious programmers will use a try: ... except: block to ensure that unblock is called even if an exception occurs when setting the button state. If it's necessary for some reason to block logging, but still execute the callback function, use block_log and unblock_log instead of block and unblock.

*2.1.3. Adopting Non-*Widgets

gtklogger's method for identifying objects in the log file only works for gtk.Widgets, because it relies on the parent/child relationship that Widgets have with their containers. However, there are other objects, derived from the gtk.Widget base class, gobject.GObject, that also emit signals that must be logged. The gtk.Adjustments that underlie scrollbars and sliders and the gtk.CellRenderers, gtk.TreeModels and gtk.TreeSelections that underlie gtk.TreeViews are examples of non-widgets that emit signals. For these to be logged, they must be *adopted* by a named Widget. Adoption establishes a pseudo-parent/child relationship between a named Widget and the non-Widget that is to be logged.

The gtklogger module provides the function adoptGObject for this purpose. For example, the following code creates a gtk.TreeView and calls the callback function selectionCB when the TreeView's selection changes:

```
tv = gtk.TreeView(...)
selection = tv.get_selection()
selection.connect('changed', selectionCB)
```

To log the changes, the `TreeView` must be named, the `TreeSelection` must be adopted, and the change signal must be logged:

```
tv = gtk.TreeView(...)
gtklogger.setWidgetName(tv, 'tv')
selection = tv.get_selection()
gtklogger.adoptGObject(selection, tv, access_method=tv.get_selection)
gtklogger.connect(selection, 'changed', selectionCB)
```

The first two arguments to `adoptGObject` are the adoptee and the adoptive parent. The third argument is a keyword argument which must be either `access_method` (specifying a member function of the parent that can be called to retrieve the adoptee) or `access_function` (specifying a non-member function[7] with the same purpose). There are two optional keyword arguments, a tuple `access_args` and a dictionary `access_kwargs`, which provide additional positional and keyword arguments to be passed to the access function.

`gtklogger` also provides a few convenience functions to facilitate the adoption process. `logScrollBars(window, name)` sets up logging for the scroll bars of the given `gtk.ScrolledWindow` object. `gtklogger.findCellRenderer` can be used as the access function when adopting a `gtk.CellRenderer` used in a `gtk.TreeView`. For details, please see the comments in the `gtklogger` source code (found in core.py and logutils.py).

2.1.4. Menus and Menu Items

`gtk.MenuItems` in `gtklogger` are handled like other widgets – they must have names and must be connected with `gtklogger.connect` – but they do have one additional requirement: submenus must be assigned to menu items with

```
gtklogger.set_submenu(menuitem, menu)
```

instead of with

```
menuitem.set_submenu(menu).
```

If a `gtk.Menu` is being used as a popup menu, `gtklogger.newTopLevelWidget` should be use to declared it to be a top level widget, and its `'deactivate'` signal should be logged with `gtklogger.connect_passive`.

2.1.5. Dialogs

Dialog boxes need special treatment in `gtklogger` because the replay mechanism needs extra clues to handle them properly. All calls to `gtk.Dialog` must therefore be replaced by `gtklogger.Dialog`. The `gtklogger` version is derived from the `gtk` version, and redefines its `__init__`, `run`, and `add_button` methods.

`gtklogger` at present does not redefine the `gtk` classes that derive from `gtk.Dialog`, such as `gtk.MessageDialog`. In order to use these classes, it will be necessary to create `gtklogger` variants of them, incorporating the `gtklogger.Dialog` modifications (from the file `gtklogger/core.py`).

2.1.6. Checkpoints

So far we've discussed four modifications that must be made to a PyGTK program in order to log and replay sessions with `gtklogger`. If the program uses threads, one more set of modifications needs to be made.

Consider a program in which a button click triggers a long calculation on a separate thread. After the calculation finishes, the user clicks on another button. If there are no other user events between the button clicks, the log file will simply contain two consecutive button click lines. While replaying (see Sec. 3.2), `gtklogger` has no way to know that the second button click shouldn't be performed until after the long calculation finishes. The thread reading the log file will execute the second button click immediately after the first, with possibly unexpected results.

The solution to synchronizing the playback of a multithread program's log file is to add *checkpoints* to the program. When recording a session, each call the application makes to `gtklogger.checkpoint(x)` inserts a line in the log file. The argument `x` is a string identifying the checkpoint. When replaying a log file, `gtklogger` won't proceed past a checkpoint in the file until the corresponding checkpoint in the program has been reached. In the example above, calling `gtklogger.checkpoint('whew')` at the end of the long calculation will insert a checkpoint into the log file between the two button-click lines, which will prevent the second click from being simulated until the calculation is done. The log file will look something like this:

```
findWidget('window:...:button1').clicked()
checkpoint whew
findWidget('window:...:button2').clicked()
```

Checkpoints have identifying labels because programs will typically contain many checkpoints and may execute many simultaneous threads. During playback, `gtklogger` keeps track of how many times, L_x, each checkpoint label x has been seen in the log file and how many times, P_x, `checkpoint(x)` has been called by the program. Each time a checkpoint x is read from the log, no further log lines will be read until $L_x \leq P_x$.

Note that, if two or more threads are both using checkpoints, this scheme is insensitive to the order in which the checkpoints are reached. It also never requires the program itself to stop and wait when it reaches a checkpoint — only the replay mechanism waits. This is important, because it means that new checkpoints can be added to a program without invalidating existing log files. The log files don't include the new checkpoints, so the replay mechanism will not wait for them. If it's necessary to add new checkpoints to an existing log file, the log file can be rerecorded (see Sec. 2.2 and 3.2).

If it's necessary to *remove* checkpoints from a program, however, it's important to update the log files. Extra checkpoints in a log file will halt the replay mechanism. If a checkpoint with a given label has been completely eliminated from a program, it's simple to remove all of its lines from a log file using a text editor. If instead, the program flow has just been modified so that a checkpoint is reached fewer times than it had been before the modification, it can be tricky to figure out which checkpoint lines need to be removed from the log file. To make this easier, calls to `checkpoint_count()` can be inserted manually into the log file. `checkpoint_count()` takes two arguments: the checkpoint label (a character string) and an integer count. The count argument is optional, and defaults to zero. `checkpoint_count` will raise an exception if the named checkpoint has not been reached in the code exactly *count* times more than it has been reached in the log file; that is, it asserts that $P_x - L_x = \text{count}$.

Checkpoints are also useful simply as a way of marking locations in log files, so that tests can be inserted at the appropriate points. See Sec. 2.3.

2.2. Recording and Replaying a Session

To begin recording a session, the program must call `gtklogger.start()`. The first argument to `start()` must be the name of the log file. The remaining three arguments are optional keyword arguments. `debugLevel` is an integer that controls debugging output (the default value is 2, which causes all log lines to be echoed to the terminal, as well as reporting signals which can't be logged for some reason). The second optional argument, `suppress_motion_events`, is discussed in Sec. 2.4.1. It can usually be omitted. The third, `logger_comments`, is a boolean that is discussed in Sec. 2.3.

Only one log file can be open for writing at a time. If `start()` is called while another file is open, the open file will be closed automatically.

To stop recording a session, the program must simply call `gtklogger.stop()`.

To load and replay a log file, the program must call `gtklogger.replay()`. This is the step that tests whether or not the program is working correctly. In the simplest case, the program can pass a test just by replaying the entire log file without crashing, raising an exception, or otherwise terminating abnormally. In most cases, however, it will be necessary to add additional tests to the log file, as described in Sec. 2.3.

`replay()` takes a number of arguments, all but one of which are optional. The required argument is the name of the log file which is to be loaded. The optional arguments are

- `beginCB`: a callback function that is called after the replay machinery is initialized, but before any lines are read from the log file. The function does not have any arguments. The default value is `None`.

- `finishCB`: a callback function that is called after the last line is read from the log file. The function does not have any arguments. The default value is `None`.

- `debugLevel`: an integer that determines the amount of debugging output. The default value is 2, with the same meaning as in the `run()` command.

- `threaded`: a boolean indicating whether or not the program uses threads. If the program is threaded, this parameter *must* be set to `True`. The default value is `False`.

- `exceptHook`: a function that is called if the replay mechanism catches an exception while executing a line from the log file. The exception is passed in as an argument.

The function should return `True` if it has handled the exception, and `False` if the exception should be propagated further. In both cases, no further lines will be read from the log file. The default value of `exceptHook` is `None`.

- `rerecord`: the name of a file in which to save the log data, just as if the session were being recorded with `gtklogger.start(rerecord)`. Re-recording a file can be a quick way of incorporating new checkpoints into an old log. When re-recording, comments and `assert` statements in the old log are copied verbatim into the new log, but the assertions are not executed.[8] The default value of `rerecord` is `None`.

- `checkpoints`: a boolean value indicating if checkpoints should be used when executing the log file. It can sometimes be useful, especially when re-recording, to ignore checkpoints. Usually, though, this causes more problems than it solves. The default value is `True`, indicating that checkpoints will be respected.

2.3. Instrumenting a Log File

After a program has been instrumented and a log file has been recorded, sometimes simply replaying the log file is sufficient to test that the program is working correctly. More often, though, it will be necessary to run some extra tests while replaying the log. This is done by adding lines to the log file, using any text editor.

`gtklogger` log files are mostly, but not quite, Python code. They are read one line at a time, and most lines are passed to the Python interpreter. Long lines can be broken by ending them with a backslash (\). One difference between a log file and a real Python script is that code blocks are not allowed in log files, because they can't be processed one line at a time. Another difference between log files and Python scripts is that log files contain special lines that begin with a few non-Python keywords, namely `checkpoint`, `pause`, and `postpone`. These lines aren't processed by the Python interpreter. `checkpoint` lines are inserted by the `gtklogger` checkpoint mechanism, as described in Sec. 2.1.6. `pause` lines, of the form `pause xxx` where `xxx` is a number, can be inserted by hand, and instruct the playback mechanism to pause for the given number of milliseconds before proceeding to the next line. (`checkpoint`s are generally much more useful than `pause`s.) `postpone` lines are inserted automatically by a few kinds of `Widget`s, and should not be modified or inserted

manually.

Like real Python scripts, log files can contain comments preceded by `#`.

The prohibition against code blocks in log files doesn't seriously limit the kinds of testing that can be done, because `import` statements *are* allowed, so a log file can contain lines like this:

```
import mytests
assert mytests.testX() == 1
```

which will raise an exception if `mytests.testX` does not return `1`. Tests should always take the form of `assert` statements – tests should generally not be run while rerecording (see Sec. 2.2), and `assert` statements in the log file are skipped in that context.

Tests should be inserted into the log file whereever it's necessary to check that the user interface or any other program component is in a particular state. They should take the form of `assert` statements. Tests should not be run while rerecording (see Sec. 2.2), and the playback mechanism skips `assert` statements in rerecord mode.

Finding the most useful locations in the log file for tests can be difficult, but some strategies are useful:

1. Add comments to the log file as it is being recorded, indicating what tests should be done. To make this easy to do, if the optional `logger_comments` argument to `gtklogger.start()` is `True`, then instead of writing the log file directly, `gtklogger` will pipe its output through a primitive GUI which allows the user to type comments into the log file. The GUI is written in PyGTK and runs in a separate process so that it won't interfere with `gtklogger`. The default value of `logger_comments` is `True`.

   In principle, anything could be added to the log file in this way, including `assert` statements. In practice, we've found it easier to instrument a log file after recording it completely.

2. Look for log lines referring to named `Widget`s. Any line that simulates an action of a particular `Widget` will begin with a call to `findWidget(...)`, where the argument is a colon-separated string containing the name of the widget and its parent `Containers`. Menu item activation corresponds to lines beginning with calls to `findMenu(...)`.

3. Use checkpoints. If the program calls `gtklogger.checkpoint()` after performing some action, the tests to make sure that the action has been performed correctly should be inserted in the log file just after the checkpoint. See 2.1.6.

4. Insert an `assert False` line in the log file, which will cause replaying to halt at that point in the file, at which time it's possible to examine the state of the interface and figure out what's happening there. This only works if the user interface is known to be working correctly, and if an `exceptHook` is being used to prevent the program from terminating with a Python `AssertionError`.

`gtklogger` contains a few routines designed to make it easier to write functions that test the state of the user interface.

- `gtklogger.findWidget(name)` returns the PyGTK `Widget` with the specified `name`, which is a colon-separated string identifying the `Widget`, as discussed in Sec. 2.1.1.

- `gtklogger.findMenu(menushell, path)` returns a `gtk.MenuItem` contained in the given `gtk.MenuShell`. `MenuShell` is a `Widget`, so it should be named and located by `gtklogger.findWidget`. `path` is a colon-separated string identifying the menu item by its name and the names of its parent menus. For example, if the `gtklogger` name of a `gtk.MenuBar` is `window:menubar`, then the `Quit` item in the `File` menu would be returned by

  ```
  findMenu(findWidget('window:menubar'), 'File:Quit')
  ```

- `gtklogger.findAllWidgets(top)` prints out the names of all named `Widgets` whose colon-separated string names begin with `top`. It won't find adopted non-`Widgets`. This is useful for debugging, and to find the full name of a `Widget` whose name you've forgotten.

2.4. Miscellaneous Functions

2.4.1. Motion Events

If a callback is connected to a `Widget`'s `motion-notify-event`, every mouse motion within the `Widget` will be logged. Often, this will create a whole blizzard of lines in

the log file, most of which aren't actually necessary. `gtklogger` contains two functions, `log_motion_events()` and `dont_log_motion_events()`, which can be called to turn motion logging on and off, so that only interesting motions are recorded. The initial status is controlled by the optional `suppress_motion_events` keyword argument to the `start()` function, whose default value is `True`.

If `log_motion_events` and `dont_log_motion_events` are passed a `gtk.Widget` argument, then they apply only to motion events within that `Widget` or its children. If they're called with no argument, they apply to *all* `Widgets`, even those whose logging was explicitly enabled or disabled by previous calls.

2.4.2. Exceptions

If a program raises exceptions that are *not* derived from the Python `Exception` base class, `gtklogger` should be notified about those exceptions by calling `gtklogger.add_exception()`. The argument to `add_exception` is the class of the expected exceptions. The function should be called before replaying log files.

2.4.3. Sanity Checking

The function `gtklogger.sanity_check(widget)` is a debugging tool that checks that the name assigned to a `Widget` correctly identifies it. If two or more `Widgets` have the same name (and their parents all have the same names), then `sanity_check` will raise an exception for at least one of them. If the given `Widget` is a `Container`, the check is run on the container and all of its children.

The function `gtklogger.comprehensive_sanity_check()` runs the sanity check on all known named `Widgets`.

3. DETAILS

This section discusses some details of the `gtklogger` implementation that aren't necessary for users to know, but are required reading for those who wish to extend or modify it.

3.1. Loggers

When a PyGTK object emits a signal, as long as that signal has been connected with `gtklogger.connect` or `gtklogger.connect_passive`, `gtklogger` creates an instance of a `GtkLogger` subclass, and uses that instance's `record` method to store the event in the log file. The `GtkLogger` class hierarchy mirrors (more or less) the PyGTK class hierarchy. If a `GtkLogger` instance can't handle a particular signal, it passes it on to its parent class's `record` method. If the signal is passed all the way back to the `GtkLogger` base class, an exception is raised.

Each `GtkLogger` subclass must have a `record()` method, taking the arguments `self`, `object`, `signal`, and `*args`. `object` is the PyGTK object that emitted the signal, and `signal` is the signal it emitted. If the signal's callback signature includes extra arguments, these are contained in `*args`. `GtkLogger.record`'s job is to return a list of strings which are to be written into the log file, one per line. These lines are Python code that, when executed by `gtklogger.replay`, will reproduce the event that triggered the signal. Those lines will have to contain information about which PyGTK object emitted the signal. To get that information, `record` should call `GtkLogger.location(object, *args)` with the same `object` and `*args` that were passed in as arguments. `location` returns a string containing the appropriate `gtklogger.findWidget` invocation for the PyGTK object.

All `GtkLogger` subclasses are actually derived from two intermediate classes, `gtklogger.widgetlogger.WidgetLogger` and `gtklogger.adopteelogger.AdopteeLogger`. These two classes take care of defining the `location` functions for all other subclasses.

Besides having `record` and `location` methods, `GtkLogger` subclasses must contain a tuple called `classes` that lists the PyGTK classes which they can log. When the `gtklogger` signal handler creates a logger for a PyGTK object, it looks for the `GtkLogger` class that lists the most derived base class for the PyGTK object. That is, the `GtkLogger` subclass that includes `gtk.Button` in its `classes` tuple will be used to log all `gtk.Button` objects and all objects in subclasses of `gtk.Button`, unless those subclasses are explicitly listed in another `GtkLogger`'s `classes` tuple.

As a concrete example, consider the simple `gtk.Button`. It's `GtkLogger` looks like this:

```
class ButtonLogger(widgetlogger.WidgetLogger):
```

```
    classes = (gtk.Button,)
    def record(self, object, signal, *args):
        if signal == 'clicked':
            return ["%s.clicked()" % self.location(object, *args)]
        return super(ButtonLogger, self).record(object, signal, *args)
```

The `record` method handles only the `'clicked'` signal. It returns a line that, when executed, will call the button's `clicked()` method, which is the programmatic equivalent of pressing the button. The call to `self.location()` generates code that ensures that the correct button is virtually clicked. Finally, in case the signal is something other than `'clicked'`, the final line calls the base class `record` function. This is why the `GtkLogger` class hierarchy mimics the PyGTK class hierarchy — signals handled in PyGTK by base classes are logged by `GtkLogger` base classes.

The `GtkLogger` for `gtk.Adjustment` objects looks like this:

```
class AdjustmentLogger(adopteelogger.AdopteeLogger):
    classes = (gtk.Adjustment,)
    def record(self, object, signal, *args):
        if signal == 'value-changed':
            return ['%s.set_value(%20.13e)' % (self.location(object, *args),
                                               object.get_value())]
        return super(AdjustmentLogger, self).record(object, signal, *args)
```

`Adjustment`s aren't `Widget`s, so their `GtkLogger` is derived from `AdopteeLogger` instead of `WidgetLogger`. Nonetheless, the same call to `self.location()` generates the code that will identify the `Adjustment` while replaying. The `'value-changed'` event will be recreated by calling the `Adjustment`'s `set_value()` method with its current value.

The `gtklogger` package only includes `GtkLogger`s for the PyGTK objects and signals that we've used in our programs[3]. These are listed in Table 3.1. Users who extend this list by adding new subclasses or new signals are invited to contribute them by sending an e-mail to the authors.

A few utility functions are provided for use in `GtkLogger`'s `record` functions.

If it's necessary for a `GtkLogger` to use a local variable in the multiline output of a `record` function, the function `loggers.localvar(basename)` should be used to acquire a unique

PyGTK object	signals
Widget	focus-in-event focus-out-event size-allocate button-press-event button-release-event motion-notify-event
Adjustment	value-changed
Button	clicked
RadioButton	clicked
ComboBox	changed
Entry	changed
Expander	activate
Paned	notify::position
MenuItem	activate
MenuShell	cancel deactivate
TreeView	button-release-event row-activated row-expanded row-collapsed
TreeSelection	changed
ListStore	row-inserted row-deleted
CellRenderer	toggled
Window	destroy delete-event configure-event

TABLE I: The PyGTK objects that `gtklogger` currently handles, and the signals that can be logged for each. The list can easily be extended by defining new `GtkLogger` subclasses.

variable name. The name will be the string given by `basename`, plus `_xxx` where `xxx` is an integer.

A line in a log file can create a `gtk.gdk.Event` object by calling `gtklogger.event()`. The first argument to `event` must be an event type, such as `gtk.gdk.BUTTON_PRESS`. The remaining arguments are keyword arguments specifying attributes of the `Event`.

3.2. The Replay Mechanism

At its core, the `gtklogger` replay mechanism is conceptually simple, but is made complicated by a few pesky details. When replaying a log file, `gtklogger` reads the file and creates a `GUILogLineRunner` instance for each line in the file. There are different kinds of `GUILogLineRunner`s for different types of lines (code, checkpoints, comments, *etc.*). `GUILogLineRunner`s have a `__call__` method that executes the line. Each `GUILogLineRunner` is installed as a PyGTK idle callback so that the log file lines can be executed without affecting the basic control loops of the program.

One complication arises in making sure that the lines are executed in sequence. When a `GUILogLineRunner` is called, it installs the *next* `GUILogLineRunner` as an idle callback, and checks to see if the *previous* line has completed. Only then does it actually execute its own code. (If the previous line hasn't completed, the current line reinstalls itself as a time-out callback, which schedules it to be run again after a specified delay, set to 100 ms by default.) It's necessary to install the next line before executing the current line, because there are situations in which a line won't complete until *after* some other PyGTK events have occurred. In particular, if a line runs code that opens a modal `Dialog` box, it may not finish until the `Dialog` is closed. However, the `Dialog` can't close without some other action on the part of the user. That action is presumably encoded in the next few lines of the log file. Therefore it's necessary to run the next lines *before* finishing the current line. (This is the reason that a program using `gtklogger` has to replace all `gtk.Dialog`s with `gtklogger.Dialog`s. The `gtklogger` version contains code that allows the replay machinery to know when it's time to run the next line of the log file.)

When a checkpoint line is encountered, `gtklogger` compares the number of times the checkpoint's label has been seen in the log file and the number of times that the same label has been passed to `checkpoint` functions in the code (as described in Sec. 2.1.6). If replaying has to wait, then the current `GUILogLineRunner` installs itself as a PyGTK time-out callback, which means that it will run again after a specified interval (100 ms, by default). It will do this repeatedly until the checkpoint is satisfied, at which point it will indicate that it has finished, allowing the next line in the log file to run.

4. EFFECTIVENESS

The `gtklogger` tool described here was developed principally to assist with testing the OOF[3] project, and all of our experience with it is in this environment.

The nature of the tests implemented using this tool is somewhat ad-hoc, as is typical of real-world software project where testing is introduced part way through the development process. Consequently, the primary benefit to the OOF project has been to ensure that the functionality for which tests have been written — necessarily that which is thought to be important by the development team — continues to pass the tests as development continues. In this way, the GUI test suite has had the character of a regression test set. In addition to ensuring that correct functionality stays correct, we have written tests for GUI bugs once they are discovered, to ensure that, once fixed, GUI bugs also stay fixed.

A more thorough testing scheme could use this tool to implement test cases generated from user feedback or from other automatic tools. Extending the GUI tests to do this is a topic of current research.

5. OBTAINING GTKLOGGER

`gtklogger` may be downloaded free of charge from

`http://www.ctcms.nist.gov/oof/gtklogger/`

That page includes the source code, this paper, and a demonstration program.

`gtklogger` was produced by NIST, an agency of the U.S. government, and by statute is not subject to copyright in the United States. Recipients of the software assume all responsibilities associated with its operation, modification and maintenance.

6. ACKNOWLEDGMENTS

Rhonald Lua and Edwin García were supported in part by the National Science Foundation through the Information Technology Research Grant DMR-0205232.

---

* Electronic address: `stephen.langer@nist.gov`

[1] URL `http://www.gtk.org`.

[2] URL `http://www.pygtk.org`.

[3] S. A. Langer, A. C. E. Reid, R. E. García, S.-I. Haan, R. C. Lua, W. C. Carter, E. R. Fuller, Jr, and A. Roosen, *Oof: Analysis of real material microstructures*, Webpage, http://www.ctcms.nist.gov/oof/index.html.

[4] J. Andersson and G. Bache, *Extreme Programming and Agile Processes in Software Engineering* (Springer Berlin/Heidelberg, 2004), vol. 3092 of *Lecture Notes in Computer Science*, chap. The Video Store Revisited Yet Again: Adventures in GUI Acceptance Testing, pp. 1–10.

[5] `gtklogger`'s widget names are not the same as the names that may be assigned by PyGTK's `gtk.Widget.set_name` function. Names set by that function are used in GTK resource files for assigning styles to widgets, and need not be unique. `gtklogger` names must uniquely identify a widget.

[6] Ideally, `gtklogger` would simply redefine the `connect` method for all PyGTK classes, but PyGTK doesn't allow this. It's necessary instead to call a `gtklogger` function explicitly.

[7] If a non-member access function is used, it must be inserted into `gtklogger`'s namespace by calling `gtklogger.replayDefine` so that it may be located when replaying.

[8] This is because log files often have to be rerecorded when something is wrong with the program, and assertions can make the program fail before the log file has been completely re-recorded.

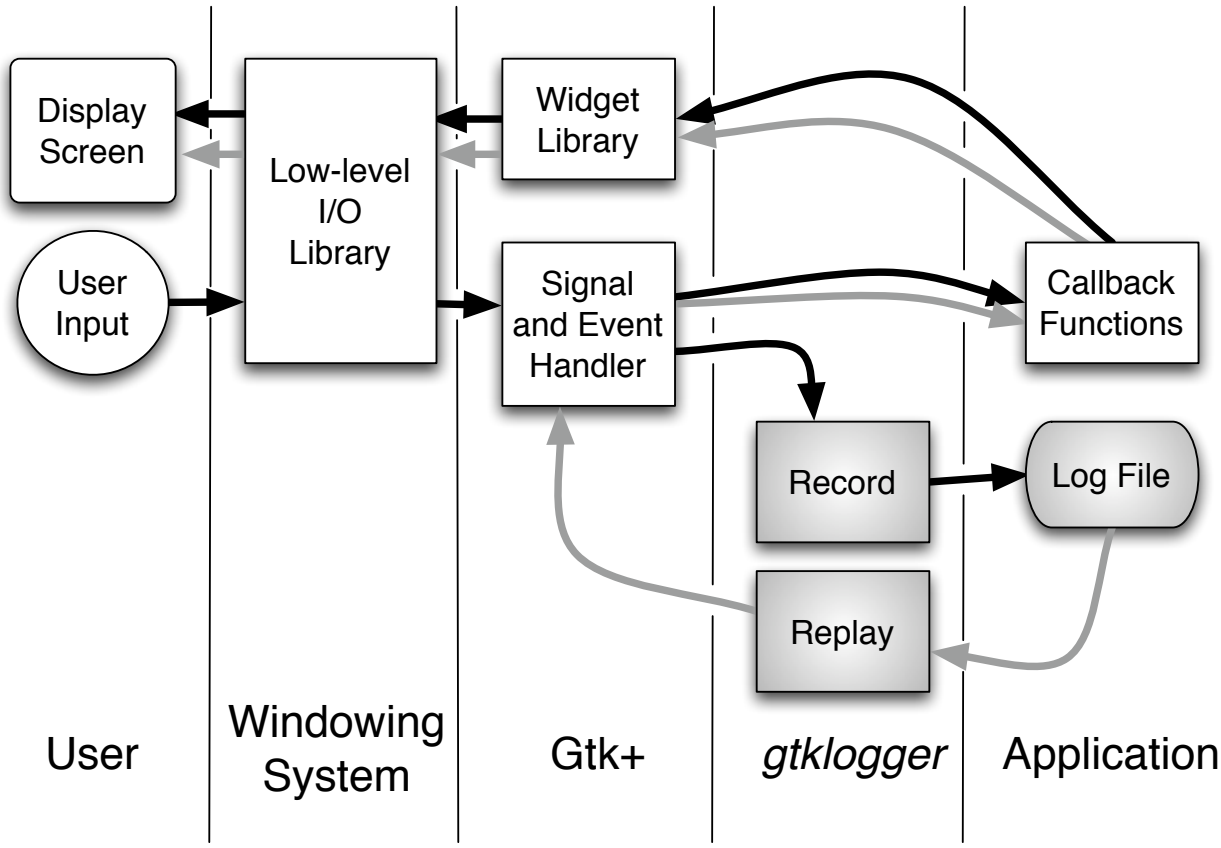

FIG. 1: The flow of control during `gtklogger` operations. Dark arrows show the control path during recording — user input is caught by the windowing system, and generates GTK+ events, which are caught both by the usual callback and by the `gtklogger` recording system, which generates an entry in a log file, but does not interfere with the execution of the callback. Light arrows illustrate the control path during playback — events are synthesized by the replay mechanism from entries in the log file, and triggers the usual chain of operations, just as though the event had originated with a user.