Testing Graphical User Interfaces

Ian Tucker

January 12, 2007

Abstract

In this document, I will be briefly looking at what a GUI is, before moving on to describe how we can test them and the problems which arise from these testing methods.

Contents

1 Introduction **3**
1.1 What is a GUI? 3

2 Background **3**

3 The Currency Conversion Program **4**
3.1 GUI Testing 101 5
3.2 Unit Testing the CC Program 7
3.3 Integration Testing the CC Program 8
3.4 System Testing the CC Program 8

4 Conclusion **12**

5 Bibliography **12**

6 Appendix **13**

1 Introduction

Graphical User Interfaces, or GUIs are all around us. Whenever you use a piece of software, chances are, you'll use a GUI. From the day to day running of your computer, to the safety critical systems of a nuclear power plant, GUIs are everywhere, and they're not going to be going anywhere soon. In this document, we will be looking at what a GUI is, and how we can test it. The rest of this document is structured as follows. GUIs will be looked at in more detail, and an example will be described. This example will be used as a basis on which we can base our examination of the three levels of testing, unit, integration and system. Then, we will briefly look at other testing methods, such as automated GUI testing. Finally, we will summarize the document. To begin, we shall lay down some definitions.

1.1 What is a GUI?

A GUI is an interface to a program. It allows the user to interact with the program using graphics, such as buttons, to represent functions in the program. GUIs are becoming more and more popular, as they allow a relatively inexperienced user to use complex pieces of software with a minimum of training. Before GUIs were invented, program interaction was done through command line based menus, where the user would type in a specific series of commands, or select options from many layers of text menus. These were very good for experience users, who could perform very complicated tasks very quickly, but training a person up to that level was very time consuming, and would often only be reached after using the software for many months or even years.

2 Background

The first GUI was created by Doug Englebart and his team at Stanford Research Institute in the late 1960s [1] in the oNLine System. The term "mouse" comes from this system as well. The fist commercial GUI based OS came from Xerox, and was called the Star, from the team at their Palo Alto Research Center. It was released to the public in 1981, but was very expensive and only sold 25 thousand. The main problem with the Star was that, even though the OS was very powerful, the computer was not, and couldn't cope, with fairly predictable results.

Steve Jobs was touring the Xerox facility and he saw the potential for these GUIs, and this is probably the place where he got his ideas from, and went to found Macintosh. Bill Gates saw the release of the Star and went to work on Windows. The Macintosh was released in 1984, while the first major GUI from Microsoft was Windows 1, released around the same time.

GUIs have come a long way since those early designs, with computers becoming more powerful, GUIs have been getting increasingly more complex. This has been a double edged sword to the computer user. While complicated GUIs mean more features in their programs, more computing power has to be devoted to making these features work. This means less power is available for other tasks. A program running on a powerful computer with a fancy GUI might take longer than the same program running on a less powerful computer, but with a very basic GUI, or no GUI at all.

Even though GUIs have changed almost beyond all recognition, the basics remain the same. They are still parts of a program, and they still have to be tested. This is what the rest of this document is focused on.

3 The Currency Conversion Program

Throughout this document, We shall be using a single GUI as an example on how to test all GUIs. The methods and principles described here can be taken and used on any GUI. This is because of the underlying nature of all GUIs is the same. They are all event driven. A GUI never does anything without someone telling it to do something. It will sit there, waiting for an input, and only then will it respond to that input in the form of any number of outputs.

The GUI that we will be using for reference is a currency conversion program's GUI.

The Currency Conversion Program

When the program is run, the user sees the above window. This is also the state the program is set to if the clear button is clicked.

The program takes an input in United States Dollars (USD) and outputs the equivalent amount in Brazilian Reals, Canadian Dollars, European Community Euros or Japanese Yen. It produces errors if there is no country or USD amount entered when the compute button is clicked, and at any time, the user can clear the GUI, which returns the program to its initial state, or close the GUI, which quits the program.

3.1 GUI Testing 101

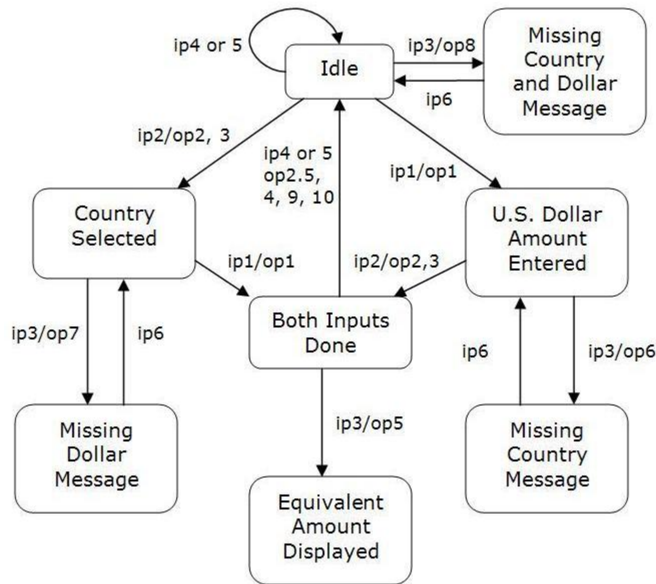
In order to unit test any GUI, we need to remember the event driven nature of the GUI. Because of this, all testing starts off by identifying all the user inputs and the visible system outputs. These are derived from the program specification, and the full table for our program can be seen in the appendix, item I.

I would like to take this opportunity to highlight part of the table which is a little artificial.

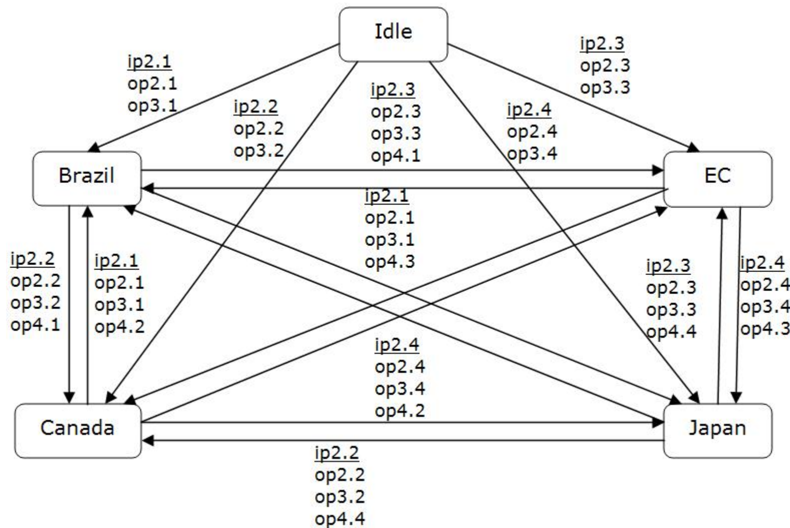
	Input Event
ip2	Click on a country button
ip2.1	Click on Brazil
ip2.2	Click on Canada
ip2.3	Click on European Community
ip2.4	Click on Japan

The input ip2 is artificial. The user cannot click on 'a country' button, he is restricted to the four choices 2.1 to 2.4. The artificial state ip2 has been introduced to make the next stage much easier. The output states op2, op3 and op4 are also artificial, for the same reason.

Once we have all the inputs an outputs, we can use a structure out of UML to make them easier to understand. Below is a basic finite state machine for the CC program



There are two things to note about this diagram. The first is that not all input events are shown. Clear and Quit can be done at any state (apart from the error states) and so have been left off for ease of understanding purposes. The second is that this is where the artificial states have come in very handy. They have allowed us to keep the complicated country select sub-graph entirely separate from the main diagram. This is what it looks like.



This entire graph would appear twice on the main graph, making it virtually unreadable, and very hard to understand. Now that we have the overall flow of control defined and laid out in an easy to understand format, we can move on to the actual testing.

3.2 Unit Testing the CC Program

The saving grace of many GUIs is that there is very little unit testing to be done. This is because most of the time, GUIs are written with public domain libraries, such as Java's Swing library, which have been thoroughly tested themselves. The means that most of the unit testing for GUIs is focused on user supplied code. In our case, these are the compute, the clear and the quit buttons. Unit testing of these can be done in two ways. You can write a test driver, as talked about earlier in this series, or you can use the GUI.

Writing a test driver may take some time, but it is the fastest way to repeat a selection of tests. For a small GUI (one developed by one person) like our one, using the GUI itself as a test bed is workable. To do this however, you need to have absolute faith in the GUI libraries. You need to be sure that values you enter are being stored correctly, and any outputs generated are being displayed correctly. If your function works perfectly, but the GUI library mistakenly displays all nines and sixes, you will be incorrectly convinced that the error is within your function.

There are other problems which arise with using the GUI as a test bed.

- Getting the test results is complicated. There is no easy way to

automate this, so a human has to be present to record the results manually.

- GUI testing done this way is generally seat-of-the-pants testing, based heavily on intuition, with little or no defined structure. This can lead to vital test routes not being followed.
- Repetition of test results is time consuming. For small GUIs, this is not too much of a problem, but for a large, more complicated GUI, exercising a set of tests can take many days.

3.3 Integration Testing the CC Program

Integration testing, like unit testing, is also quite trivial with small GUIs. The idea of integration testing is to make sure that the methods are interacting with each other properly.

The amount of integration testing that needs to be done depends heavily on the way the program was implemented. In our example, there are two different ways in which the program could be implemented. The first has one method, the compute method, and the country selected is chosen by a switch statement, with the equivalent amount being calculated there and then, and passed to the GUI. The second way is a lot more modularized, with each country button calling it's own method to adjust a global exchange rate when it is clicked. The compute method is a lot smaller, and it just calculates the equivalent amount based on the exchange rate variable.

The first of these has very little integration testing, as all the code is done in one module. The second has a bit more integration testing, as you have to make sure that each of the button's code is passing the correct exchange rate, and the compute function handles it correctly.

3.4 System Testing the CC Program

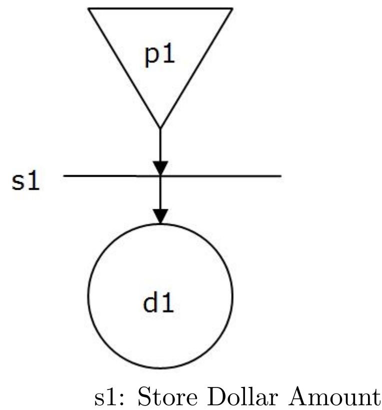
As we have seen, unit and integration testing of most small GUIs is trivial. This is very handy, because system testing of even a small GUI is very long and complicated, and large GUIs (GUIs developed by a team of people) can be a nightmare to test thoroughly. Luckily for us, we have a few tools available to us that make the process much more manageable.

The main aim of system testing is to produce a set of test threads, which, when exercised, test every route through the program. To help us do this, we have Event Driven Petri-Nets (EDPNs). These are used to describe Atomic System Functions (ASFs), which are a collection of one user input, and one or more system outputs, with the possibility of data inputs and outputs too.

In order to construct these EDPNs we need to compile a list of port input and port output events, a list of ASFs and a list of Data places.

These are very similar to the original list of inputs and outputs, and can be found in the appendix, item II. Unlike the input and output event lists, these have no artificial events for selection of countries. This is because they are not needed here, and would only complicate matters. We can also compile a list of ASFs, these are at appendix III. The data places list is at appendix IV, with one consideration. The data place 'Country Selected', while acceptable, is not accurate to the program. The more correct way would be to define 4 separate data places, one for each country, but for our purposes, the one artificial data place makes things easier, and doesn't detract from our testing completeness. This is thanks to the public domain libraries that the GUI has been built from, and as these handle the selection of the country, it is not something we have to worry about in too much detail.

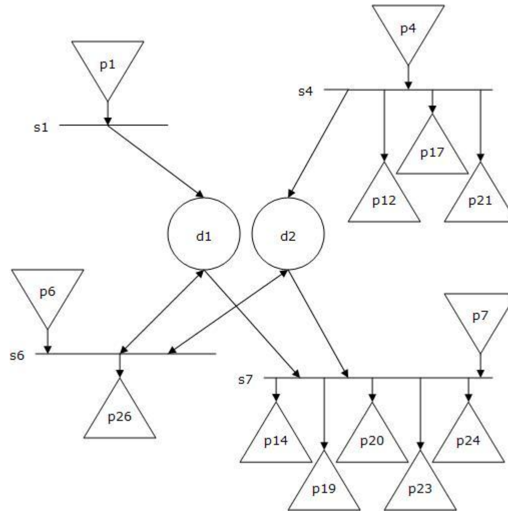
Once we have these four lists defined, we can use them to construct our EDPNs of the GUI. A simple one is shown below



s1: Store Dollar Amount

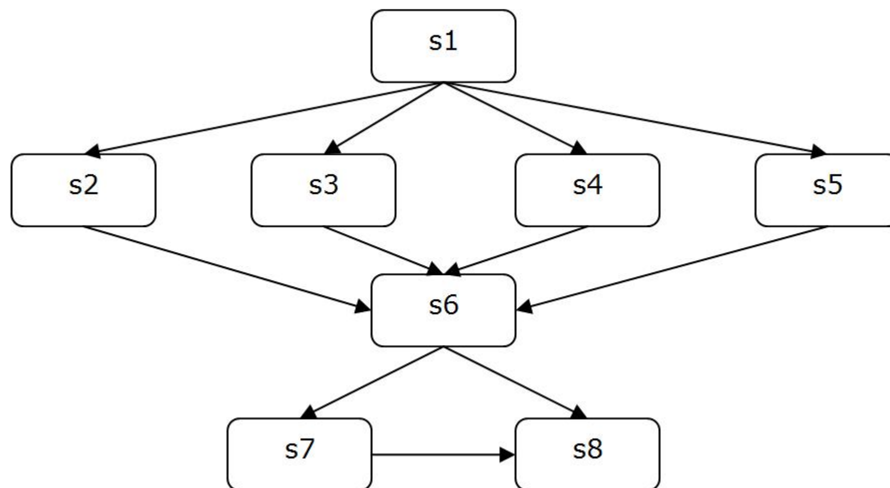
The ASF (the line) cannot be executed to produce outputs until all the inputs have been done. This is usually controlled by a separate side graph showing the flow of control, but this has been left out for simplicity's sake. In the above example, s1 will only be executed once the user has input a USD amount. The output is to store the amount in the data place d1.

The greatest strength of EDPNs is that they can be combined into bigger, more complicated petri-nets. We can therefore combine lots of small petri-nets into a large petri-net describing a system thread. One such petri-net is shown below.



EDPN Composition of four ASFs into a System Level Thread

This graph is quite complicated, and while we could use this method to create one massive EDPN for the entire system, it would be almost unreadable. Fortunately, our EDPN is still made up of ASFs, so we can make a graph of ASF sequences to do the same job. Below is a partial graph of the system.



Partial Directed graph of mainline ASF Sequences

This represents exactly half of the graph of the system, the other half start by selecting a country before entering a USD amount.

The other thing to note is that the graph is directed, so loops are not allowed. In reality, loops do exist, and the ramifications of this will

be discussed later.

We can now finally talk about how we test a GUI. The first though is to test all ASFs individually. This is a very low level of testing, and is not a good idea, for two reasons.

1. Not all ASFs have visible system level outputs
2. Some (s1) might not even have any port outputs

If you were testing s1 on it's own, there is no easy way to check that they amount has been correctly stored.

Thus, we have to come up with a new idea. We know that a thread is a route through the system, so what about exercising a suitable set of threads. This idea brings one questing into mind. How do you define a 'suitable' set of threads?

Obviously, since we want to test our GUI thoroughly, we should at the very least construct a set of threads that tests every ASF, every port input event and every port output event. Testing every ASF means that we test every port input and output by definition, you can't have a port input which doesn't trigger at least one ASF, and you can't have a port output that isn't used by at least one ASF.

If we refer back to the graph of ASF sequences, we can see that if we construct the set of threads T [T1, T2, T3, T4], where T1 = [s1, s4, s6, s7], T2 = [s1, s2, s6, s7] T3 = [s3, s1, s6, s7] and T4 = [s5, s1, s7, s8], we can see that exercising this set of threads will give us complete coverage of all the ASFs, and therefore all port inputs and port outputs as well.

This is a good start, but we can make some improvements. The set of threads T5 = [s1, s2, s6, s3, s6, s4, s6, s5, s6, s7, s8] tries to test some next-level user behavior. In this set, the user converts one amount to all 4 currencies, clears and quits. This would be a good set to test, as many users might use the program in this way.

A user might change his mind about the currency he wishes to convert to, so a set of threads T6 = [s1, s2, s3, s4, s5, s6, s7, s8] might be appropriate, where the user goes through all the countries, and finally settles on Yen to convert.

We can go on. Perhaps a user is just bored, and wants to play around for a bit with the program. He might do something like T7 = [s1, s2, s3, s2, s3, s2, s3, s8], toggling between two countries, then quitting the program. These examples raise the important question on when do we stop.

Clearly the set of test thread-sets is infinite, as a user can toggle between two countries for all eternity. Obviously, we cannot test them all, so, like any other program, we are reduced to testing as intelligently as possible. With GUIs, however, we can do something about that.

A GUI can control the user's actions on itself, by only allowing the user to do one thing at once. Sections of the GUI can be disabled and

enabled, forcing the user down a strict path. This makes testing a lot easier, as the number of paths is reduced to a small number, but greatly restricts the useability of the program. If our GUI forced the user to enter a USD amount before doing anything else, then only allowing them to click on the compute button after selecting a country, the number of possible paths through the system is reduced to the much more manageable amount 4, but the user will get very annoyed and will never use the program again.

4 Conclusion

There is a fine balance to be struck between useability and testability, and in the world where the consumer is king, testability gets a very small priority. Companies want their software to be the easiest to use, with the most features, and thoroughly testing those programs, let alone their GUIs, is something that far too often falls by the wayside. Fully automated testing of programs and GUIs is still in its infancy, and the packages to do it can cost thousands of pounds, something which most small and many medium companies just can't afford.

To sum up, GUI testing is possible, and there are defined methods to carry it out, but in the race to publish software as quickly and as cheaply as possible, it is rarely done to the extend that it could be. This isn't much more than a minor inconvenience to a home user, but a nuclear power plant is a different story.

5 Bibliography

References

[1] Press, Larry, "Personal computing: Windows, DOS and the MAC," *Commun. ACM,* **33**(11), (New York, NY, USA: ACM Press. 1990): 19–26.

6 Appendix

I

	Input Event
ip1	Enter U.S. dollar amount
ip2	Click on a country button
ip2.1	Click on Brazil
ip2.2	Click on Canada
ip2.3	Click on European Community
ip2.4	Click on Japan
ip3	Click on Compute Button
ip4	Click on Clear Button
ip5	Click on Quit Button
ip6	Click on OK in error message
	Output Event
op1	Display U.S. dollar amount
op2	Display currency name
op2.1	Display Brazilian reals
op2.2	Display Canadian dollars
op2.3	Display European Community euros
op2.4	Display Japanese Yen
op2.5	Display ellipsis
op3	Indicate selected country
op3.1	Indicate Brazil
op3.2	Indicate Canada
op3.3	Indicate European Community
op3.4	Indicate Japan
op4	Reset selected country
op4.1	Reset Brazil
op4.2	Reset Canada
op4.3	Reset European Community
op4.4	Reset Japan
op5	Display foreign currency value
op6	Error msg: must select a country
op7	Error msg: must enter USD amount
op8	Error msg: must select country and enter USD amount
op9	Reset U.S. Dollar amount
op10	Reset equivalent currency amount

II

	Port Input Events
p1	Enter U.S. dollar amount
p2	Click on Brazil
p3	Click on Canada
p4	Click on European Community
p5	Click on Japan
p6	Click on Compute Button
p7	Click on Clear Button
p8	Click on Quit Button

	Port Output Events
p9	Display US Dollar Amount
p10	Display Brazilian reals
p11	Display Canadian dollars
p12	Display E.U. euros
p13	Display Japanese Yen
p14	Display ellipsis
p15	Indicate Brazil
p16	Indicate Canada
p17	Indicate European Community
p18	Indicate Japan
p19	Reset Canada, E.U., Japan
p20	Reset Brazil, E.U., Japan
p21	Reset Brazil, Canada, Japan
p22	Reset Brazil, Canada, E.U.
p23	Reset U.S. Dollar amount
p24	Reset equivalent currency amount
p25	End Application
p26	Display equivalent currency amount

III

	Atomic System Fuctions
s1	US Dollar Amount Stored
s2	Sense Click on Brazil
s3	Sense Click on Canada
s4	Sense Click on EU
s5	Sense Click on Japan
s6	Sense Click on Compute Button
s7	Sense Click on Clear Button
s8	Sense Click on Quit Button

IV

	Data Places
d1	US Dollar amount entered
d2	County selected