Testing Graphical User Interfaces

Jaymie Strecker
University of Maryland, USA

Atif M Memon
University of Maryland, USA

INTRODUCTION

In recent years, an emerging trend in software products has been toward the use of graphical user interfaces (GUIs). More user-friendly than traditional, text-based interfaces, GUIs serve as the front-end for a large portion of today's software applications. Technologies like Ajax are helping to spread familiar GUI interaction styles to Web applications. With the rise of ubiquitous computing, users are interacting with GUIs in a widening range of situations—not just with their PCs, but with their dishwashers and cars. Critical applications, such as banking systems, are moving to GUIs as well. Thus, quality assurance for GUI-based software is growing more important every day.

With GUIs, users enjoy many degrees of freedom in the way they interact with the software. While this benefits users, it challenges testers. Because users may interact with a GUI in a variety of unexpected ways, it is difficult to insure that the software meets its functional requirements (correctness) and non-functional requirements (e.g., usability) for all possible interactions. The difficulties are compounded by the frequent intersection of GUIs with other emerging technologies, including component-based and service-oriented architectures. New trends in software development, such as rapid development cycles, globally distributed developers, and open-source projects, make the quality assurance process ever more challenging.

This chapter describes the state of the art in testing GUI-based software. Traditionally, GUI testing has been performed manually or semimanually, with the aid of capture-replay tools. Since this process may be too slow and ineffective to meet the demands of today's developers and users, recent research in GUI testing has pushed toward automation. Model-based approaches are being used to generate and execute test cases, implement test oracles, and perform regression testing of GUIs automatically. This chapter shows how research to date has addressed the difficulties of testing GUIs in today's rapidly evolving technological world, and it points to the many challenges that lie ahead.

BACKGROUND

A GUI provides a visual front-end through which a user can interact with a software application. Although there are various models for GUI design, the most commonly used in practice and in software-testing research—and hence the model assumed in this chapter—is the WIMP model with *w*indows, *i*cons, *m*enus, and *p*ointing devices (Nielsen, 1993). The GUI is made up of *widgets*—such as buttons, text boxes, and labels—that the user can manipulate to send input to the underlying software and the software can, in turn, manipulate to send output to the user. Each widget has a set of *properties*—for example, "font", "width", "enabled"—each of which has some *value*—for example, "Helvetica", "100", "true" (Yuan & Memon, 2007).

Widgets are contained in *windows*, which may either be *modal* or *modeless*. A modal window blocks the user's interaction with other windows while it is active, whereas a modeless window imposes no such restrictions. A *window's state* at any particular time is the set of all triples (w,p,v) such that w is a widget in the window, p is a property of w, and v is the value of p. The *GUI state* then consists of the state of all windows in the GUI (Yuan & Memon, 2007).

As the user interacts with the GUI, the state of both the GUI and the underlying software can change. When the user performs an *event* on the GUI—such as clicking a button or typing in a text box—a piece of application code called an *event handler* is executed. The event is the basic unit of interaction with a GUI. To accomplish a task, a user typically must perform multiple events in sequence. Hence, a *GUI test case* consists of a sequence of events (Yuan & Memon, 2007).

Several tools and techniques are available to aid testing of GUI-based applications, varying greatly in the level of automation they provide. Ignoring the GUI altogether, test harnesses like JUnit can interact directly with the underlying software much like the GUI would. However, this may require major changes to the GUI's architecture, and, at any rate, it leaves an important part of the end-user software untested.

JUnit has been extended in tools such as JFCUnit, Pounder, and Jemmy Module to interact with the application under test through its GUI. With these tools, test cases must be written manually. Alternatively, a tester can generate test cases by recording sequences of events, which the tester manually performs on the GUI, using a capture-replay tool. Some capture-replay tools—for example, CAPBAK and TestWorks—record events in terms of mouse coordinates, while others—for example, WinRunner, Abbot, and Rational Robot—record the GUI widgets associated with events. The latter are more robust in the face of superficial changes to the GUI layout (Memon & Xie, 2005).

All of the tools and techniques mentioned so far automate the execution of test cases but still require substantial effort on the part of the tester to generate test cases, define the test oracle, and modify the test suite as the application under test evolves. Tools like the visual test-development environment created by Ostrand, Anodide, Foster, and Goradia (1998) streamline the testing process but do not depart from the conceptualization of GUI testing as a fundamentally manual process. Similarly, while Kasik and George (1996) have shown how genetic algorithms can be used to augment a test suite, they leave much work to the tester. Fortunately, new techniques based on various types of models of the GUI are shifting much of the burden of the testing process from humans to machines.

The most popular type of GUI model, the state-machine model, makes it possible to generate test cases—or perform model-checking, a related activity—automatically (Belli, 2001; Berstel, Reghizzi, Roussel, & Pietro, 2005; Dwyer, Carr, & Hines, 1997; Holzmann & Smith, 1999; Shehady & Siewiorek, 1997; White & Almezen, 2000). But techniques based on state-machine models have serious drawbacks. These techniques require that the model be created manually, that a formal specification be written, or that the source code be annotated—in any case, a potentially laborious task susceptible to human error. Further, since the effectiveness of the test cases generated from the state-machine model depends on the model creator's definition of "state", two testers testing the same application may get quite different results (Yuan & Memon, 2007). Techniques for generating test cases from UML diagrams suffer from similar weaknesses (Vieira, Leduc, Hasling, Subramanyan, & Kazmeier, 2006).

Rather than modeling a GUI in terms of states, others have modeled it in terms of events. Memon, Pollack, and Soffa (2001) have used automated planning to generate test cases that consist of sequences of events chosen to accomplish tasks specified by the tester. In this approach, model creation requires substantial human effort: although the events in the model are identified automatically, their preconditions and effects must be defined manually. More recently, techniques have used event-based models to further reduce the amount of effort required in the testing process while improving its effectiveness. These are described in the next section.

GUI TESTING WITH EVENT-FLOW MODELS

Events are central to the dynamic structure of a GUI-based application. A user accomplishes tasks via the GUI by performing sequences of events. Thus, the execution of the application occurs as the execution of a sequence of event handlers, each of which may depend on and may also affect the state of the application. Users may interact with the application in unexpected ways, so the event handlers may be executed in unexpected orderings. In these respects, GUI-based applications differ from traditional, batch-style software (e.g., compilers), which receives some input, processes it, produces some output, and terminates. Traditional testing techniques like code-based coverage criteria that were designed for such software may not work as well for much differently-structured GUI-based applications, so new techniques have been developed to address GUIs' event-driven nature (Memon, 2002).

The previous section showed how GUI-testing tools and techniques have evolved to be faster and more effective. Notable advances have been achieved through model-based testing, using various types of models. In recent years, one type of model has proved particularly successful: the *event-flow graph*.

Event-Flow Graph

In an event-flow graph, a GUI is represented by a graph whose vertices represent events and whose edges represent the *follows* relationship. Event e_1 is said to *follow* event e_2 if e_1 can be executed immediately after e_2, with no events intervening. Test cases can be generated rapidly and automatically by traversing the EFG, and coverage criteria can be defined in terms of the EFG. Variations of the EFG have been used to further improve the cost-effectiveness of GUI testing. Each of these topics will be elaborated upon after the process of creating an EFG is explained (Xie & Memon, 2006).

An EFG can be reverse engineered semi-automatically from a GUI in a process called *GUI ripping*. A single GUI window is ripped by identifying and recording properties of all of the widgets it contains, then executing any events available in the window that open new windows. This can be accomplished by running the GUI with reflection to access the currently open windows and inspect their widgets. Widgets likely to open new windows can be identified based on conventions in GUI design: clicking on a widget whose caption ends in "..." typically opens a window. As