Automated Test Oracles for GUIs

Atif M. Memon*
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
atif@cs.pitt.edu

Martha E. Pollack†
Dept. of Computer Science
and Intelligent Systems
Program
University of Pittsburgh
Pittsburgh, PA 15260
pollack@cs.pitt.edu

Mary Lou Soffa‡
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
soffa@cs.pitt.edu

ABSTRACT

Graphical User Interfaces (GUIs) are critical components of today's software. Because GUIs have different characteristics than traditional software, conventional testing techniques do not apply to GUI software. In previous work, we presented an approach to generate GUI test cases, which take the form of sequences of actions. In this paper we develop a test oracle technique to determine if a GUI behaves as expected for a given test case. Our oracle uses a formal model of a GUI, expressed as sets of objects, object properties, and actions. Given the formal model and a test case, our oracle automatically derives the expected state for every action in the test case. We represent the actual state of an executing GUI in terms of objects and their properties derived from the GUI's execution. Using the actual state acquired from an execution monitor, our oracle automatically compares the expected and actual states after each action to verify the correctness of the GUI for the test case. We implemented the oracle as a component in our GUI testing system, called Planning Assisted Tester for grapHical user interface Systems (PATHS), which is based on AI planning. We experimentally evaluated the practicality and effectiveness of our oracle technique and report on the results of experiments to test and verify the behavior of our version of the Microsoft WordPad's GUI.

Keywords

GUI testing, GUI Test Oracles, Automated Oracles.

*Partially supported by the Andrew Mellon Pre-doctoral Fellowship.

†Partially supported by the Air Force Office of Scientific Research (F49620-98-1-0436) and NSF (IRI-9619579). Effective Sep 1, 2000: Department of Electrical Engineering and Computer Science, University of Michigan. pollackm@eecs.umich.edu

‡Partially supported by NSF (CCR 9808590 and EIA 9806525).

1. INTRODUCTION

Graphical User Interfaces (GUIs) are critically important components of most current software [11]. As with all software, the behavior of a GUI, as well as the underlying code, needs to undergo extensive testing to help ensure that it behaves correctly. Although extensive research has been devoted to testing conventional software, the resulting techniques and approaches are not applicable when testing GUIs, because GUIs have special characteristics. Thus, testing technology for GUIs requires new approaches. In a previous paper, we described an approach to automatically generate test cases, which are sequences of actions, for GUIs by using Artificial Intelligence planning techniques [9]. In this paper, we focus on the problem of *automatically* determining, given a test case, whether a GUI behaves correctly.

The characteristics of GUIs present special challenges when verifying a GUI's behavior [12, 10, 24]. Many of these challenges stem from the fact that GUIs are event-based systems. With conventional software, a test case usually consists of a single set of inputs, and the expected result is the output that results from completely processing that input. The form of the output can be readily specified, e.g., as the values of a certain set of variables. With GUIs, the input is an entire action sequence, where the effect of each action may depend upon the effects of its previous actions. There is no specific output: rather, each action affects the state of the GUI. Moreover, comparison of the expected and actual GUI states cannot wait until the entire action sequence has been executed. Instead, it is necessary to verify the state of the GUI after the execution of each action; otherwise, incorrect GUI behavior for one action may result in a state in which future actions in the sequence cannot be executed at all.

The above challenges suggest the need to develop an automated oracle that answers the question of whether a GUI executing under a test case behaves as expected. The automation should occur both in the derivation of the expected states and the comparison of the expected and actual states. The development of an automated test oracle for GUIs has certain requirements. First, we need a way of modeling the GUI's intended behavior so that we can automatically derive its *expected state* during the execution of a test case. In order to model the GUI's intended behavior, we need to develop a representation of the GUI elements and actions. Second, we need to represent the state of the executing GUI in a form that is suitable for comparison with the expected

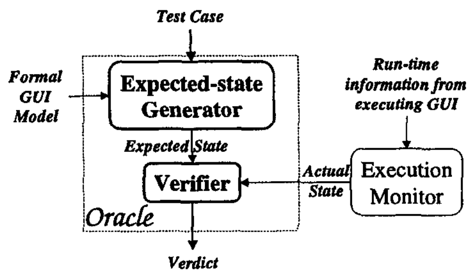

Figure 1: An Overview of the GUI Oracle.

state description. Finally, we need to design a mechanism to automatically compare the expected state with the state of the executing GUI.

In this paper, we present a technique to develop an automated GUI test oracle. An overview of the oracle is shown in Figure 1. The oracle uses a formal model that is developed by the oracle designer from the GUI specifications. The model is composed of the GUI objects and a set of properties for those objects. GUI actions are represented in the model by their preconditions and effects. The oracle automatically derives the expected state using the model and the actions from a test case. Likewise, the actual state is also described by a set of objects and properties typically found in a GUI toolkit or specialized GUI language. The oracle obtains the actual state information from an execution monitor. A verifier in the oracle then automatically compares the two states and determines if the GUI is performing as expected. We implemented our technique in our GUI testing system PATHS (the Planning Assisted Tester for grapHical user interface Systems), and show how we were able to facilitate automation of the GUI test oracle by exploiting the AI planning-based tools already present in PATHS. We experimentally evaluated the oracle on a version of Microsoft Word Pad and provide timing results that establish the feasibility of our approach.

In particular, the important contributions of the method presented in this paper include the following.

- We define a formal model of a GUI derived from specifications that is useful in testing. In this paper we demonstrate its usefulness in developing oracles.
- Our oracle is general in that it will work for any GUI as long as an appropriate model can be established. The oracle is also portable across platforms since it depends on properties that can be acquired from GUI toolkits or special programming language features.
- The technique allows reuse of operator definitions that commonly appear across GUIs. These definitions can be maintained in a library and reused to help develop oracles for GUIs.
- We show our oracle creation process as a natural extension of our already implemented planning-based test-case generation system. We reuse the planning operators defined for test-case generation and apply them in a unique way to create oracles.

In the next section, we describe our GUI model. In Section 3, we show how this model is used to determine the expected state sequence of the GUI for a test case. In Section 4, we show how to compare the expected state information with the executing GUI's actual state. In Section 5, we demonstrate how the oracle is used in testing an example GUI. Section 6 describes our implementation and presents experimental results. We present related work in Section 7 and concluding remarks in Section 8.

2. MODELING THE GUI

We begin by describing how a GUI can be formally modeled, and then show how that model can be used to compute expected states of the GUI.

2.1 Objects and Properties

We model a GUI as a set of objects, (window, menu, button, text, etc.), a set of properties of those objects (background color, font, is-open, etc.), and a set of actions that change the properties of certain objects (set-background-color, etc.). Each GUI will use certain types of objects with associated properties; at any specific point in time, the GUI can be described in terms of the specific objects, or GUI elements that it currently contains, and the current values of their properties.

More formally, we model a GUI at a particular time t as:

- its **objects** $O = \{o_1, o_2, \ldots, o_m\}$, i.e., the objects the GUI currently contains, and
- the **properties** $P = \{p_1, p_2, \ldots, p_l\}$ of those objects. Each property p_i is an n_i-ary Boolean relation, for $n_i \geq 1$, where the first argument is an object $o_1 \in O$. If $n_i > 1$, the last argument may be either an object or a property value, and all the intermediate arguments are objects. The property value is a constant drawn from a set associated with the property in question: for instance, the property "background-color" has an associated set of values, {white, yellow, pink, etc.}. We assume a distinguished set of properties, the *object types*, which are unary relations, e.g., "window" or "button".

Thus we might specify the state of a (extraordinarily simple) GUI at some particular time by noting that it currently has two window objects, `w17` and `w29`, for which the following properties hold: `window(w17)`, `window(w29)`, `background-color(w17, red)`, `is-current(w17)`. The *state* of a GUI at a particular time is everything that is currently true of it. So a description of the state would contain information about the types of *all* the objects currently extant in the GUI, as well as *all* of the properties of each of those objects.

There are several points that should be noted about our description of properties. First, properties are relations, not functions, and so there may sometimes be multiple values for the same property of a given object. For example, there may be multiple objects in a window. Next, properties as we have defined them are *fluents* [8], i.e., relations which are true in some situations (or states of the world) and not others. An everyday example of a fluent is the relation `president(US, Clinton)`, with the obvious meaning, where the state it is evaluated in is the state of the real world. Our fluents are evaluated with respect to a state of the GUI. Finally, note that a fluent may be undefined in some states, for example,

president(US, Dole) in the state of the world in the year 1567, or **background-color(w24, blue)** in the state of a GUI immediately after window **w24** has been destroyed.

In practice, we can determine the set of object types and properties for our GUI model in several different ways. One approach would be manual examination of the GUI: we look at it, and write down all the object types and properties we can discover. This approach is prone to incompleteness, especially since GUIs may have hidden properties that must be checked during verification. For example, the *tab order* of windows in a GUI (the order in which windows receive input focus when the **Tab** key is pressed) is a property that is not visible. A second approach is to derive the objects and properties directly from the GUI's specifications, which will describe them either directly or implicitly within the descriptions of GUI actions. A third approach is to examine the language or toolkit used to develop a particular GUI. For example, if the GUI was developed using the Java language, then the GUI objects would be instances of the **swing** GUI components of the Java swing package, and the properties would correspond to the instance variables (also called data members in C++) of each object. Visual programming environments provide a more direct interface to properties. For example, Borland's C++ Builder presents the properties as a table for the currently selected object.

The third approach can lead to a larger set of object types and properties than does the second. This is because the set of object types and properties made available by a language or toolkit may not all be used in the construction of a particular GUI. For example, one might use Borland's C++ builder to construct a simple GUI in which the user is not permitted to manipulate the text color, and in which the text color does not influence the execution of any other action. (In fact, Microsoft's NotePad is like this.) Thus, if one establishes the set of properties from the GUI's specifications, text color will not be amongst the properties modeled, whereas if one establishes it from the toolkit used for development, text color will be included as a property in the model. We thus distinguish between the *complete set* of properties for a GUI, which are all those that would be identified by our third (language/toolkit-based) approach, and the *reduced set*, which includes only those that would be identified by our second (specifications-based) approach. Note that the reduced set is always a (possibly improper) subset of the complete set of properties.

2.2 Actions

The state of a GUI is not static; actions are used to change it over time. We model actions as state transducers, i.e., we define an action as follows:

Definition: The **actions** $A = \{a_1, a_2, \ldots, a_n\}$ associated with a GUI are functions from one state of the GUI to another state of the GUI. □

Actions may be parameterized, e.g., **set-background-color(w, x)**. Whenever the action **set-background-color(w19, yellow)** is executed in a state in which window **w19** is open, the background color of **w19** should become **yellow** (or stay **yellow** if it already was), and no other properties of the world should change. This example illustrates that, typically, actions can only be executed in some states; **set-background-color(w19, yellow)** cannot be executed when window **w19** is not open.

We use the notation $s_j = [s_i, a]$ to denote that s_j is the state resulting from the execution of action a in state s_i. We can string actions together into sequences. We will say that $a_1; a_2; \ldots; a_n$ is a *legal action sequence for initial state* s_0 iff there exists a sequence of states, $s_0; s_1, \ldots; s_n$ such that $s_i = [s_{i-1}, a_i]$ for $i = 1, \ldots, n$. Extending the notation above, we use $s_j = [s_i, a_1; a_2; \ldots a_n]$, where $a_1; a_2; \ldots; a_n$ is a legal action sequence, to denote that s_j is the state that results from executing the specified sequence of actions starting in state s_i.

Definition: A **GUI test case** is a pair $< s_0, a_1; a_2; \ldots a_n >$, consisting of an initial state and a legal sequence of actions for that state. □

We model actions using their descriptions in the GUI specifications: after all, the purpose of verification is to ensure that the implementation of the actions matches the expected behavior promised in the specifications. In the next section, we provide further details about modeling actions.

3. DERIVING EXPECTED STATE

We can now see how the model of the GUI can in principle be used to determine the expected state of a GUI after the complete or partial execution of any test case. Recall that actions are modeled as state transducers. For any test case $< s_0, a_1; a_2; \ldots a_n >$, the sequence of states $s_1; s_2; \ldots s_n$ such that $s_i = [s_{i-1}, a_i]$ for $i = 1, \ldots, n$ represents the expected state of the GUI after each action is executed, starting in s_0. The question is how, in practice, to compute these expected states.

It is of course infeasible to give exhaustive specifications of the state mapping for each action: in principle, as there is no limit to the number of objects a GUI can contain at any point in time, there can be infinitely many states of the GUI.[1] Thus, we adopt the technique of modeling GUI actions using *operators*, which specify their preconditions and effects:

Definition: An **operator** is a 3-tuple <Name, Preconditions, Effects> where:

- **Name** identifies an action and its parameters.
- **Preconditions** is a set of positive ground literals[2] $p(arg_1, \ldots, arg_n)$, where p is an n-ary property (i.e., $p \in P$).
- **Effects** is also a set of positive or negative ground literals $p(arg_1, \ldots, arg_n)$, where p is an n-ary property (i.e., $p \in P$).

□

[1]Of course in practice, there are memory limits on the machine on which the GUI is running, and hence only finitely many states actually possible, but the number of possible states will be extremely large.

[2]A literal is a sentence without conjunction, disjunction or implication; a literal is ground when all of its arguments are bound; and a positive literal is one that is not negated. It is straightforward to generalize the account given here to handle partially instantiated literals. However, it needlessly complicates the presentation for this paper.

We write *Pre*(Op) and *Eff*(Op) to represent the set of preconditions and effects, respectively, for operator Op. An operator is applicable in any state s_i in which all the literals in *Pre*(Op) are true. In the resulting state s_j, all of the positive literals in *Eff*(Op) will be true, as will all the literals that were true in s_i except for those that appear as negative literals in *Eff*(Op). The scheme for encoding operators we use is the same as what is standardly used in the AI planning literature [14, 22, 23]; the persistence assumption built into the method for computing the result state is called the STRIPS assumption. A complete formal semantics for operators making the STRIPS assumption has been developed by Lifschitz [7].

The STRIPS-style of encoding operators also makes it fairly easy to derive result state $s_j = [s_i, a]$, via simple additions and deletions to the list of relations representing state s_i.

For example, if we were to define an operator for the `set-background-color` action, then we would get the following operator definition:

> **Name:** `set-background-color(wX: window, Col: Color)`
> **Preconditions:** `is-current(wX), background-color(wX, oldCol), oldCol` $\neq$ `Col`
> **Effects:** `background-color(wX, Col)`

Going back to our simple example of the GUI in which the following properties were true: `window(w17), window(w29), background-color(w17, red), is-current(w17)`. If we applied the above operator, with variables bound as `set-background-color( w17, blue )`, we would get the following state: `window(w17), window(w29), background-color(w17, blue), is-current(w17)`, i.e., the background color of window `w17` would change from `red` to `blue`.

The next state is obtained from the current state S_c and the operator's effects e as follows:

1. Delete all literals in S_c that unify with a negated literal in e, and
2. add all positive literals in e.

Thus, using a formal model of a GUI, we can derive the expected state, given an initial state and a sequence of actions.

Given that GUI specifications can describe the intended behavior of actions in terms of their preconditions and effects [5, 4], it is relatively straightforward for the test designer to construct operators for the GUI model. In fact, as we will see later, the operators can also be used in other aspects of testing.

4. STATE COMPARISON

We have just described how to model a GUI and use that model to derive the expected state. Now we turn to the question of how to compare that information to the actual state.

The simplest approach is manual comparison. One manually executes a test case, and after each step, manually compares the appearance of the GUI with the expected state at that time. Manual verification has at least two problems: (1) it is labor intensive, and (2) often the GUI state includes "hidden" properties that are not visually accessible.

Our goal is therefore to automate the process of extracting actual GUI state information in a form that is suitable for comparison with the expected state description. We define an *execution monitor* to be a process that, given an executing GUI, returns the current values of all the properties in the complete set for the GUI. Once the actual values of properties for an element or elements are known, the verifier can compare them against the expected values, to determine if they are equal. We, therefore define the verifier to be a process that compares the expected state of the GUI with the actual state and returns a *verdict* of equal or not equal.

The remaining question, then, is what properties should be compared during the verification process. There are several possible answers to this question, and the decision amongst them establishes the *level of testing* performed:

Changed-Properties Verification: Here, comparison is made only for those properties that were expected to change as a result of the immediately preceding action. That is, if action a was just executed, only the properties that are included in *Eff*(a) are compared against their expected values. Although efficient, this level of testing will fail to detect changes to properties that change when they are not expected to change. For example, if the background color of a window changes, but it was not expected to change, the error would go unnoticed.

Relevant-Properties Verification: Here, all the properties in the reduced property set (see Section 2.1 above) are checked. Recall that the reduced property set includes all the properties that the current GUI is ever supposed to access. This is, thus, a much more extensive level of testing than changed-properties verification, but it may still fail when some GUI property P changed in the executing GUI, but P was not a part of the GUI specification. For example, consider a GUI for a plain-text editor, e.g., MS NotePad in which users cannot change the text color. If some action in the test case has the unintended effect of changing the text color, then this error would go unnoticed, since the color information was not encoded in the expected state.

Complete-Properties Verification: Here, a check is made for all the properties that a language or toolkit provides for a GUI. Recall that the verifier has access to the complete set of properties. The only problem is the absence of an expected state to compare against all these additional properties. The currently available expected state encodes only the reduced property set. To address this problem, before the test case is executed, a baseline *complete expected state* of the GUI is created. During test-case execution, the comparisons are done between the GUI's actual state and the updated complete expected state.

In practice, the test designer can choose a combination of the above levels of testing. For example, the verifier can perform changed-properties verification after each test action and complete-properties verification after every 10 actions.

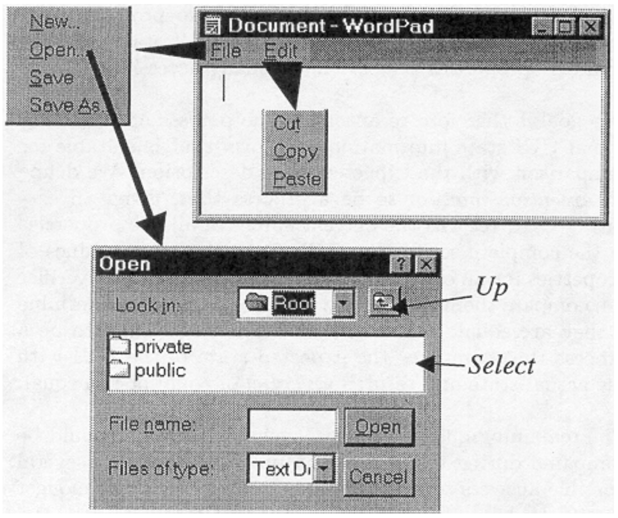

Figure 2: The Example GUI.

We now have all the necessary mechanisms to develop an automated test oracle for GUIs.

5. A GUI EXAMPLE

In this section we show, through an example, how a GUI is tested using an automated test oracle.

Figure 2 presents a small part of the Microsoft WordPad's GUI. This GUI can be used for loading text from files, manipulating the text (by cutting and pasting) and then saving the text in another file. At the highest level, the GUI has a pull-down menu with two actions (`File` and `Edit`). The GUI user can execute the GUI actions to make other elements available. For example clicking on `File` opens a menu with `New`, `Open`, `Save` and `SaveAs` actions. `Edit` opens a menu with `Cut`, `Copy`, and `Paste` actions. `Open` and `SaveAs` open windows with several more actions. These actions are used to traverse the directory hierarchy and select a file. The up button moves up one level in the directory hierarchy and clicking on files and directories is used to select files or enter subdirectories respectively. The window is closed by clicking on either `Open` or `Cancel`.

We assume that the GUI's test cases are given. Recall that we defined a test case as a pair $(S_0, a_1; a_2; a_3; ...; a_n)$, where S_0 is the initial state and $a_1; a_2; a_3; ...; a_n$ is an action sequence. Consider, for example, the sequence of actions to be applied to our version of the WordPad software shown in Figure 3. This sequence of actions transforms the GUI from the initial state S_0 shown in Figure 4(a) to the one shown in 4(b). Figure 4(a) shows a collection of files stored in a directory hierarchy. When the actions are executed on the GUI, the new document shown in Figure 4(b) is created and then stored in file *f4.doc* in the `/Root/Latex/Samples` directory.

5.1 The Oracle Designer

To test the above GUI, an *Oracle Designer* uses the GUI specifications to develop a formal model of the GUI. The

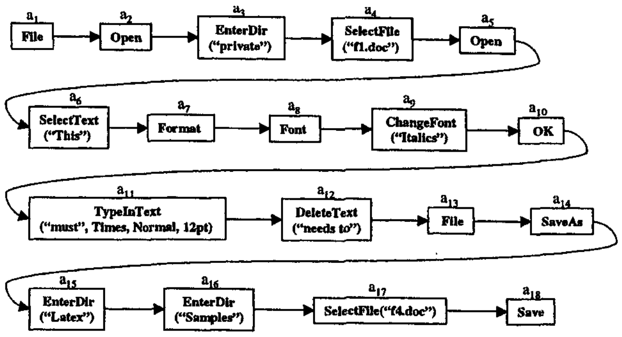

Figure 3: An Action Sequence for our Version of the WordPad Software

Property	Args	Semantics
in	*File*, *Text*	*File* contains *Text*
contains	*ParentDir*, *Dir*	*ParentDir* contains *Dir*
containsfile	*Dir*, *File*	*Dir* contains *File*
currentFile	*File*	The current file is *File*
currentFont	*Font*, *Style*, *Size*	The current font is *Font*, style is *Style*, and size is *Size*
font	*Text*, *Font*, *Style*, *Size*	*Text* is in *Font*, *Style*, and *Size*
isCurrent	*Dir*	*Dir* is the current directory
onScreen	*Text*	*Text* is displayed on the screen
selectedFile	*File*	*File* is selected
selectedText	*Text*	*Text* is highlighted

Table 1: Some Properties, their Parameters, and Semantics.

rest of the process, i.e., deriving an expected state sequence for each test case, executing the test case, extracting the actual state, and verifying its outcome of the test case is handled automatically.

The first step in deriving the expected state is for the oracle designer to use the GUI specifications to identify the properties of the elements of the GUI. The semantics of some properties used in this example are shown in Table 1. The columns show the property name, the parameters, and the semantics of each property. The oracle designer then represents the initial state (Figure 4) in terms of the identified properties as shown in Figure 5. The initial state describes the file structure (using the properties `contains()` and `containsFile()`), and the contents of the file *f1.doc* using the property `in()`. Additional properties are used to describe the fonts, current file, and the current directory.

By using the actions described in the specifications, the oracle designer defines the preconditions and effects of the operators. Figure 6 shows an example of an operator called `Open`,

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.