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).

This paper appears in the Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8), held in San Diego, CA, on Nov. 6-10, 2000.

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

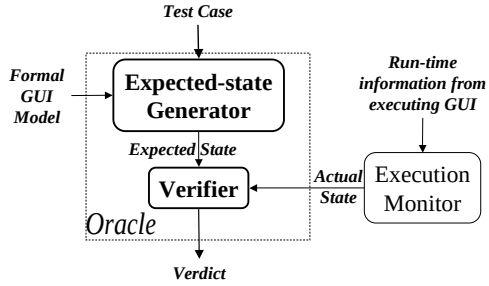

Figure 1: An Overview of the GUI Oracle.

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 $s_j = [s_i, a_1; a_2; ...a_n]$, 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.

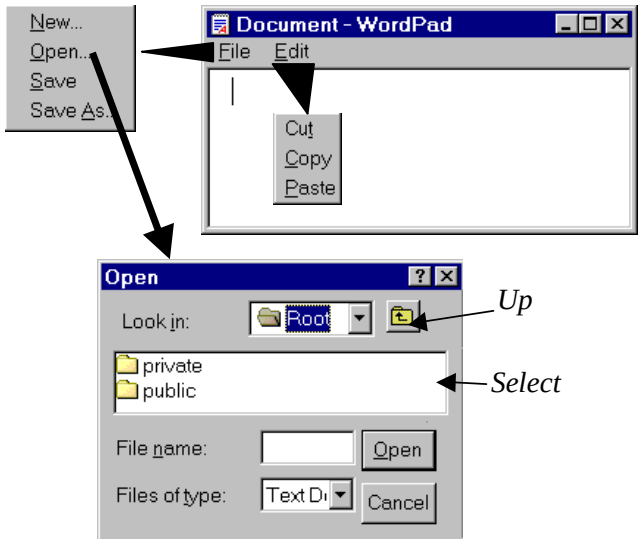

Figure 2: The Example GUI.

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

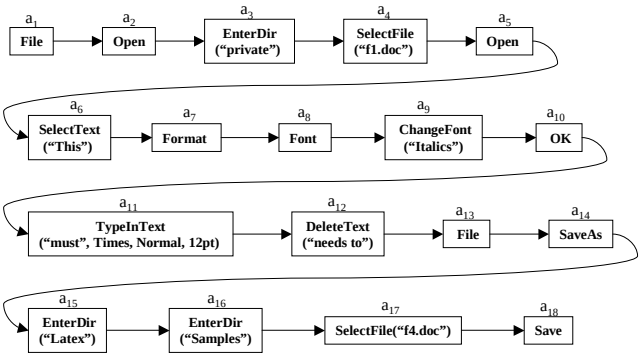

Figure 3: An Action Sequence for our Version of the WordPad Software

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`,

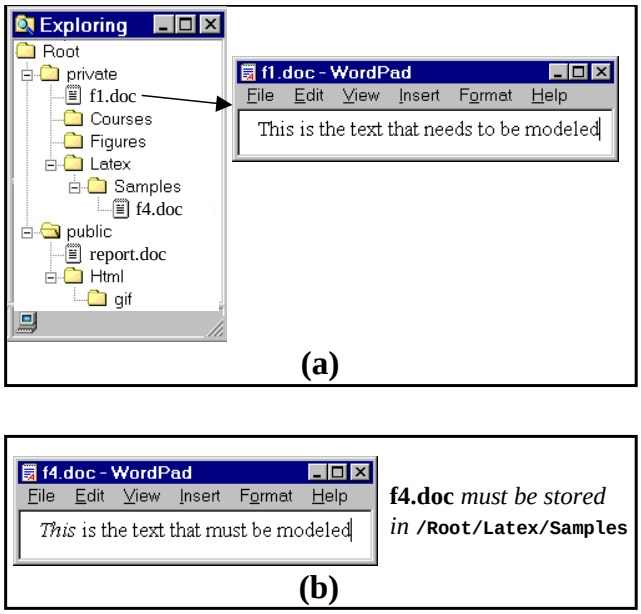

Figure 4: The Action Sequence of Figure 3 Transforms the GUI from: (a) the Initial State, to (b) the Final State

representing the `Open` action from the `File` menu. The operator `Open` takes two parameters, `dir` and `file`. The operator is available only if its precondition, `containsfile(dir, file)` is `TRUE`, i.e., directory "`dir`" contains the file "`file`". The effects of applying this operator are that the `currentFile` value is modified, all the objects on the screen are deleted, and all the objects in the file are displayed on the screen. Quantifiers and conditional statements are used to make the notation concise and intuitive. They are later replaced with their expansions when the expected state is derived.

5.2 The Automated Oracle

Using the operators defined by the oracle designer, the automated oracle derives the GUI's expected state corresponding to the given test case. The expected state sequence is derived from S_0 by using the method outlined in the previous section. The next expected state is automatically obtained by applying a_1 on S_0, i.e., $S_1 = [S_0, a_1]$. The process is repeated until the entire expected state sequence has been derived. For example, consider the expected state shown in terms of properties for actions a_4 and a_5 in Figure 7. (Note that the shown subsequence is a part of the sequence shown in Figure 3) The expected state corresponding to a_4 is represented as S_4. The GUI's state changes after action a_5 (`Open`) is executed. The new state obtained is S_5. The changes are highlighted using bold font. As mentioned earlier in the description of the `Open` operator, the `currentFile` value changes, and the objects from the file are now displayed on the screen (using the property `onScreen()`).

The test case and expected state sequence shown in Figure 7 have all the necessary components to carry out a successful test run and can be used for manual testing. The tester alternates between the test-case actions and the expected

initial:
contains(root private)
contains(private Latex)
contains(Latex Samples)

containsfile(Samples f4.doc)
containsfile(private f1.doc)

currentFont(Times Normal 12pt)

in(f1.doc "**This**")
font("**This**" Times Normal 12pt)

in(f1.doc "**is the**")
font("**is the**" Times Normal 12pt)

in(f1.doc "**text**")
font("**text**" Times Normal 12pt)

in(f1.doc "**that**")
font("**that**" Times Normal 12pt)

in(f1.doc "**needs to**")
font("**needs to**" Times Normal 12pt)

in(f1.doc "**be modeled**.")
font("**be modeled**" Times Normal 12pt)

Figure 5: Representing the Initial State.

Operator Name
Open(`dir`: DIRS, `file`: FILES)
Preconditions
containsfile(`dir, file`)
Effects
/ The current file is now* `file` **/*
currentFile(`file`)
/ Now there are no objects on the screen */*
¬onScreen(`obj`) ∀ `obj` ∈ OBJECTS
/ All objects in* `file` *are now on the screen */*
onScreen(`obj`) ∀ `obj` ∈ OBJECTS | in(`file`, `obj`)

Figure 6: An Operator.

state, executing the input events in the test case and checking the GUI state by verifying each property.[3] However, we have fully automated test execution by implementing the execution monitor and the verifier.

Now that the expected state has been automatically derived, it is compared with the actual state. The actual state of the executing GUI is obtained from the execution monitor, which maintains a list of all the properties of our version of the WordPad software. At each step in the test case, the verifier uses the values of all these properties to check them for correctness. Thus, in our example, the expected state shown in S_4 and S_5 will be automatically compared with the actual GUI state when the test case is executed.

6. IMPLEMENTATION

In this section, we first give an algorithm that shows how the components of the test oracle are used when testing the GUI. We also show the details of how the expected state is derived from the current state. Then we describe an implementation of our oracle and results of experiments to determine the time needed to derive the expected state and execute the verifier and execution monitor.

[3]Note that since the expected state has been derived from the specifications the names of properties may not match those in the toolkit. Renaming of properties may be needed at this step to match those used in the toolkit.

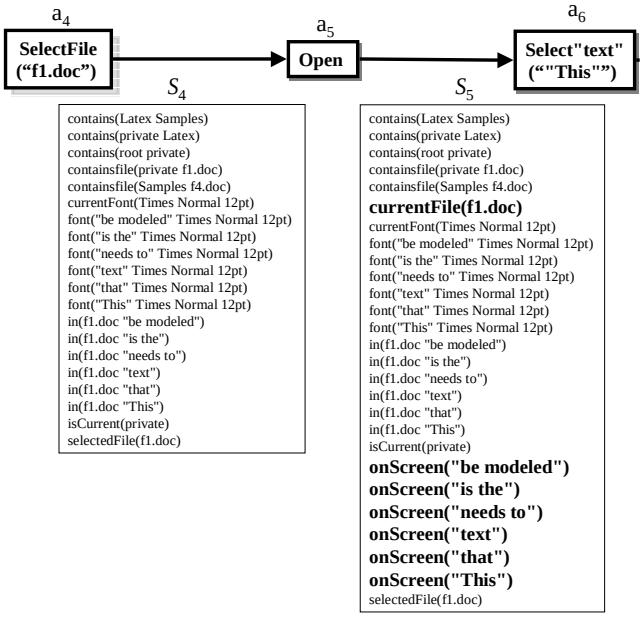

Figure 7: A Few Test-Case Actions with Expected State Information.

6.1 GUI Testing Algorithm

Figure 8 gives a high-level view of the main testing algorithm (`TestGUI`) and a procedure `ExpStateGen`, invoked by `TestGUI`. The algorithm `TestGUI` executes a test case automatically on the GUI, examining its actual state and comparing it with the expected state. The algorithm takes three parameters: (1) the `levelOfTesting`, which determines what properties will be compared by the verifier, (2) the test case `T` to be executed on the GUI; `T` contains the expected initial state and a sequence of actions, and (3) the operators `GUI_Operators` representing the abstract model of the GUI. Note that each action in the test case has a corresponding definition in `GUI_Operators`. The algorithm returns a verdict, depending on the outcome of the test case execution. For each action in the test case, `TestGUI` calls the procedure `ExpStateGen` (line 9) to determine the expected state of the GUI. If `ExpStateGen` is successful, then the action in the test case is automatically executed (line 12) on the GUI and its actual state is determined by invoking the execution monitor `ExecMonitor` (line 13). Both the expected and actual state are compared by the verifier (line 15) that performs comparisons based on the current level of testing. `TestGUI` returns the verdict (line 31), i.e., the outcome of the execution of the test case.

The procedure `ExpStateGen` takes as input the current state of the GUI (`currentState`), the `action` to be executed on the GUI, and the GUI (`operators`). Every action in the test case has a corresponding operator definition (line 38). The action contains the actual parameters of the operator definition, which are substituted for the formal parameters (line 39). `ExpStateGen` performs an extra check to determine if the preconditions of the operator are satisfied in the current state (lines 40..42). If they are not satisfied, then there is an error in the test case, and this result is propagated to the calling procedure. If the preconditions are satisfied, the new state is computed by applying the effects of the operator. If the effects contain a negated property, then it is deleted from the new state (lines 45..46) and if it contains a positive property, it is inserted (lines 47..48) in the new state. The result `newState` is returned to the calling algorithm.

6.2 Implementation

In an earlier paper we presented the design of a test case generation system based on AI plan generation techniques that used **planning operators** for GUIs [9]. In the current research, we leverage off our planning-based approach to create test oracles for GUIs by essentially reusing the planning operators used for test case generation.[4] We implemented the expected-state generator, execution monitor, and the verifier. We have incorporated the GUI test oracles into our GUI testing system – PATHS. Figure 9 shows an overview of our testing system in terms of its components, and the flow of information. The GUI specifications are used to create a formal model of the GUI as well as to implement the GUI. The GUI model is used to create test cases and corresponding expected state. The test cases are executed on the GUI, and the oracle verifies the behavior of the GUI for the test case.

We implemented the expected-state generator in C, running under Linux. The expected-state generator produces the expected states of all the test cases offline, during test case generation. As each test case is generated, the expected state generator uses the operators to produce the corresponding expected state.

We implemented the execution monitor and verifier in Borland C++ Builder, running under Windows NT.[5] In designing an execution monitor, we maintained a list of all the properties of the executing GUI and extracted the values after each action of the test case. Some properties were visible, e.g., open menus, so we could retrieve their values from the screen by using a process called *screen scraping*,[6] but other properties required getting values from the executing GUI's state by using function calls.

Implementing the verifier was straightforward. We chose to perform *relevant properties verification*. During comparison, we checked for equivalence of the expected and actual states.

We also implemented an automated test-execution system, so that all the test cases could be automatically executed without human intervention. Automatically executing the test cases involved generating the physical mouse/keyboard events. Since our test cases are represented at a high level of

[4]Note that during test case generation, we make use of hierarchical planning, and hence derive hierarchical operators. For test oracles, we restrict ourselves to primitive operators, i.e., those that directly correspond to GUI actions.

[5]Our current implementation of the test case generator and expected-state generator runs under Unix because the planner that we use is Unix-based. We execute test cases on Windows NT because we are using the Windows API to generate mouse movements and keyboard events.

[6]Screen scraping is a traditional way to selectively remove information from a host application's terminal interface for reuse. Typically, the information is programmatically accessed.

```
1  ALGORITHM: TestGUI(
2  levelOfTesting,  /* changed, relevant, or complete property
3                                                  verification */
4  T,                          /* test case {S0, a1;a2;a3;…;an} */
5  GUI_Operators               /* {Op1, OP2, OP3, ..., OPn}. Each Opi =
6                                 <Name, Preconditions, Effects>*/ ) {
7     State ← S0;
8     foreach action a in <a1, a2, a3, …, an> {
9        expState ← ExpStateGen(State, a, GUI_Operators);
10       if (expState == TEST_CASE_INVALID)
11          break;
12       ExecuteAction(a, GUI); /* Automatically execute action on GUI */
13       actualState ← ExecMonitor(GUI);
14            /* check actual state and expected for this LEVEL_OF_TESTING. */
15       if (Verifier(expState, actualState,
16                          levelOfTesting) == FALSE)
17             break;
18       State ← expState;}
19
20 if (TEST_CASE_INVALID) {
21    error("Invalid Test Case");
22    debugInfo("Actual GUI State = ", actualState);
23    debugInfo("Expected GUI State = ", expState);
24    Verdict = INVALID;}
25 if (FALSE) {/* if verifier reported FALSE, then GUI is incorrect*/
26    report("GUI failed the test case");
27    debugInfo("Actual GUI State = ", actualState);
28    debugInfo("Expected GUI State = ", expState);
29    Verdict = INCORRECT;}
30 else Verdict = CORRECT;
31 return(Verdict);}
32
33 PROCEDURE: ExpStateGen(
34 currentState,     /* properties, {p1, p2, p3, ..., pn} - the state of the GUI */
35 action,           /* step of the test case  --  actionName(parameters)*/
36 operators         /* {Op1, OP2, OP3, ..., OPn}. */) {
37
38 opDef ← Lookup(action, operators); /* get operator for action */
39 op ← Bind(opDef, action); /* bind all variables in op def. */
40 p ← preconditions(op); /* extract the preconditions of the operator */
41 if (Satisfied(p, currentState) == FAILED)
42    return(TEST_CASE_INVALID);
43 e ← effects(op); /*extract the effects of the operator*/
44 newState ← currentState;
45 foreach (f in e) {/*delete all properties that are negated in effects*/
46    if (negated(f)) delete f from newState;
47 foreach (f in e) {/*insert all properties that are positive in effects*/
48    if (positive(f)) insert f in newState;
49 return(newState);}
```

Figure 8: The GUI Testing Algorithm.

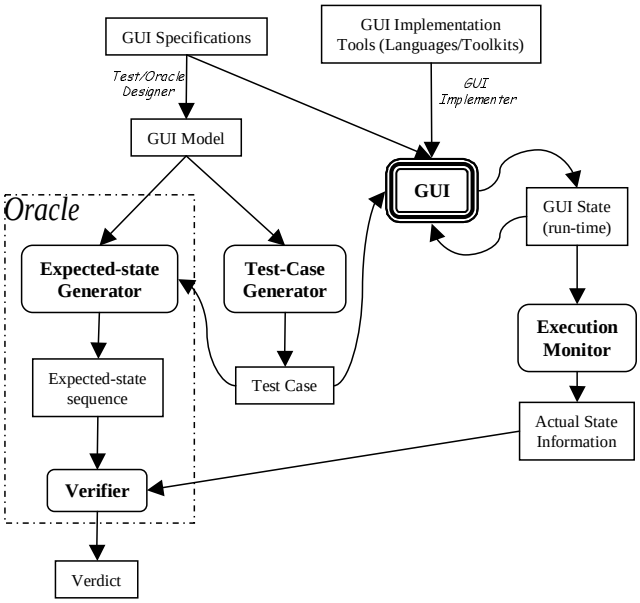

Figure 9: A Tool for GUI Testing.

abstraction, we translate the high-level actions into physical events. The actual screen coordinates of the buttons, menus, etc. were derived from the layout information.

6.3 Experimental Results

To explore the practicality of our approach, we evaluated the performance of the oracle on our WordPad GUI. We wanted to determine (1) the execution time to derive the expected state information, and (2) the time to execute the verifier and the execution monitor. In both cases, we compared the times with test case generation and execution time to determine the extra time needed to derive the expected state and execute the verifier and the execution monitor.

Our experiments are designed to help determine the scalability of our expected-state generator and test-oracle executor. We generated 290 test cases of lengths varying from 6 to 56 actions. All test cases were generated for, and executed on, our version of the WordPad software. This software consists of 36 modal windows containing a total of 362 actions (not counting short-cuts). Our version of WordPad is more or less similar to Microsoft's WordPad except for the **Help** Menu, which we did not model.

For our first experiment, we implemented our test case and expected-state generator in C. We executed our system on a Pentium-based computer (350MHz, 256MB RAM) running Linux.

The results of this experiment are summarized in Figure 10. The x-axis shows the test case length and the y-axis shows the time in seconds. As the graph shows, the significant portion of the time was spent in generating the test cases. The expected state was derived much faster. Note that the total time needed to generate the test cases and expected state was very small. In fact, we could generate all of our 290 test cases and their corresponding expected state sequences

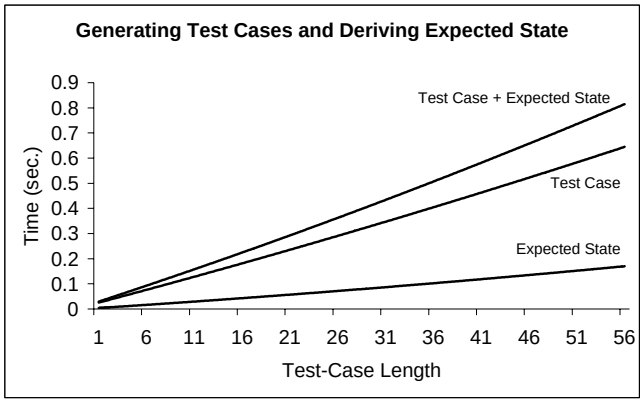

Figure 10: Time needed to Generate the Test Cases and Expected-State Information.

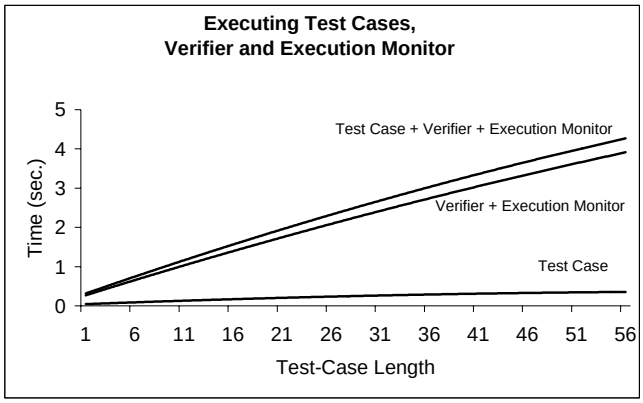

Figure 11: Time needed to Execute the Test Cases and Verifier.

in a total of 75.84 sec. CPU time.

Next, we implemented the verifier, execution monitor, and an automated test-execution system using Borland's C++ Builder. We executed these on a Pentium-based machine (350MHz, 256MB RAM) running Windows NT. We performed relevant-properties verification, i.e., we checked all the properties in the expected state after each action of the test case. Note that we deliberately chose to perform this more expensive level of testing to determine the worst-case time for oracle execution. As seen in Figure 11, the total time needed to execute the verifier and the execution monitor was very small. All 290 test cases required less than a total of 10 minutes to execute.

These experiments demonstrate that the use of planning for test case generation and oracle creation can result in an efficient testing paradigm. The human effort required to create the test oracle was reasonable and the effort can be amortized by using the same GUI model for test case generation.

7. RELATED WORK

To our knowledge, no work has been done in automating test oracles for GUIs. However, the need for such an oracle had been stressed in the literature. Ostrand et al. present a visual Test Development Environment (TDE) [13] for testing GUIs. They indicate the need to develop a facility for defining result comparison actions in test scenarios, which will give the test designer the ability to augment test scripts with oracles to check the state of the GUI as well as the system state and computation results. Shehady et al. present a FSM based technique to generate test cases for GUIs. Once the test cases have been generated, the expected output sequences can be determined by applying the test cases to the FSM model and recording the outputs [18]. They, however, do not use the FSM state to verify the GUI's behavior.

There has been some work done to create oracles for conventional software. Few techniques to generate the expected state have been developed. In most cases the expected behavior of the software is assumed to be provided by the test designer. The expected behavior is specified by the test designer in the form of a table of pairs *(actual output, expected output)* [15], or as temporal constraints that specify conditions that must not be violated during software execution [16, 1, 2, 17], or as logical expressions to be satisfied by the software [3]. This expected behavior is then used by the verifier by either performing a table lookup [15], FSM creation [6, 2], or boolean formula evaluation [3] to determine the correctness of the actual output.

Richardson in TAOS (Testing with Analysis and Oracle Support) [16] proposes several levels of test oracle support. One level of test oracle support is given by the `Range-checker` which checks for ranges of values of variables during test-case execution. A higher level of support is given by the `GIL` and `RTIL` languages in which the test designer must specify temporal properties of the software.

Siepmann et al. in their TOBAC system [19] assume that the expected output is specified by the test designer and provide seven ways of automatically comparing the expected output to the software's actual output.

A popular alternative to manually specifying the expected output is by performing reference testing [20, 21]. Actual outputs are recorded the first time the software is executed. The recorded outputs are later used as expected output for regression testing.

8. CONCLUSIONS

In this paper, we presented a new technique to develop an automated GUI test oracle. The test oracle automatically derives the expected state sequences and compares the actual and expected states after each action in the test case. The oracle generates the expected state sequences from a formal model developed by the test/oracle designer using the GUI specifications. The GUI model contains operators, representing GUI actions in terms of their preconditions and effects. The oracle obtains the actual state from an execution monitor. The actual state is represented as a set of objects and properties. The oracle then compares the two states and determines if the GUI is performing as expected.

We have demonstrated that our technique can be both practical and useful by deriving expected state sequences for our version of the Microsoft WordPad software's GUI and us-

ing them to test the software's GUI. Our experiments have shown that we can generate and execute a large number of test cases automatically in very little time.

One of the tasks currently performed by the test/oracle designer is the definition of the preconditions and effects of the operators. Such definitions of commonly used operators can be maintained in libraries, making this task easier. We are also currently investigating how to automatically generate the preconditions and effects of the operators from a GUI's specifications.

9. REFERENCES

[1] L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21 of *ACM Software Engineering Notes*, pages 106–117, New York, Oct.16–18 1996. ACM Press.

[2] L. K. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 140–153, Dec. 1994.

[3] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *Proceedings of the 21st International Conference on Software Engineering*, pages 267–276. ACM Press, May 1999.

[4] M. Frank, J. J. G. de, D. Gieskens, and J. D. Foley. Building user interfaces interactively using pre- and postconditions. In *Proceedings of CHI '92*, 1992.

[5] M. Green. *The Design of Graphical User Interfaces*. Ph.d. thesis, Department of Computer Science, University of Toronto, 1985.

[6] L. J. Jagadeesan, A. Porter, C. Puchol, J. C. Ramming, and L. G. Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 525–537, Berlin - Heidelberg - New York, May 1997. Springer.

[7] V. Lifschitz. On the semantics of STRIPS. In M. P. Georgeff and A. L. Lansky, editors, *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, Timberline, Oregon, June-July 1986. Morgan Kaufmann.

[8] J. McCarthy. Situations, actions, and causal laws. Memo 2, Stanford University Artificial Intelligence Project, Stanford, California, 1963.

[9] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for GUIs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 257–266. ACM Press, May 1999.

[10] B. A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CS-93-183, Carnegie Mellon University, School of Computer Science, July 1993.

[11] B. A. Myers, J. D. Hollan, and I. F. Cruz. Strategic directions in human-computer interaction. *ACM Computing Surveys*, 28(4):794–809, Dec. 1996.

[12] B. A. Myers, D. R. Olsen, Jr., and J. G. Bonar. User interface tools. In *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems – Adjunct Proceedings*, Tutorials, page 239, 1993.

[13] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for GUI systems. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98)*, volume 23,2 of *ACM Software Engineering Notes*, pages 82–92, New York, Mar.2–5 1998. ACM Press.

[14] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of KR'89*, Toronto, Canada, pp 324-331, May 1989.

[15] D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In T. Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 58–65, 1994.

[16] D. J. Richardson. TAOS: Testing with analysis and oracle support. In T. Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA): August 17–19, 1994, Seattle, Washington, USA*, ACM Sigsoft, pages 138–153, New York, NY 10036, USA, 1994. ACM Press.

[17] D. J. Richardson, S. Leif-Aha, and T. O. OMalley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, May 1992.

[18] R. K. Shehady and D. P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 80–88, Washington - Brussels - Tokyo, June 1997. IEEE Press.

[19] E. Siepman and A. R. Newton. TOBAC: Test Case Browser for Object-Oriented Sofwtare. In *Proc. International Symposium on Software Testing and Analysis*, pages 154–168, New York, Aug. 1994. ACM Press.

[20] J. Su and P. R. Ritter. Experience in testing the Motif interface. *IEEE Software*, 8(2):26–33, Mar. 1991.

[21] P. Vogel. An integrated general purpose automated test environment. In T. Ostrand and E. Weyuker, editors, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 61–69, New York, NY, USA, June 1993. ACM Press.

[22] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[23] D. S. Weld. Recent advances in AI planning. *AI Magazine*, 20(1):55–64, Spring 1999.

[24] W. I. Wittel, Jr. and T. G. Lewis. Integrating the MVC paradigm into an object-oriented framework to accelerate GUI application development. Technical Report 91-60-06, Department of Computer Science, Oregon State University, 1991.