Challenges and Opportunities for Improving Code-based Testing of Graphical User Interfaces

Marcel R. Karam*   Sergiu M. Dascalu   Rami H. Hazimé***

* American University of Beirut, Lebanon
Department of Computer Science
11 Bliss St., P.O. Box 11-3246
E-mail: marcel.karam@aub.edu.lb
Phone: +961-1-350 000 ext. 4234

** University of Nevada, Reno, USA
Department of Computer Science and Engineering
1664 N. Virginia St., MS 171
Reno, NV, 89523, USA
E-mail: dascalus@cse.unr.edu
Phone: +1-775-784 4613
Fax: +1-775-784-1877

Corresponding Author: Sergiu M. Dascalu, University of Nevada, Reno,
Department of Computer Science and Engineering
1664 N. Virginia St., MS 171, Reno, NV, 89523, USA
E-mail: dascalus@cse.unr.edu, Phone: +1-775-784 4613
Fax: +1-775-784-1877

Challenges and Opportunities for Improving Code-based Testing of Graphical User Interfaces

Marcel R. Karam* ***Sergiu M. Dascalu***** ***Rami H. Hazimé****

*American University of Beirut, Lebanon
**University of Nevada, Reno, USA

Abstract

The research presented in this paper introduces an execution model for graphical user interfaces (GUIs) that we have developed and formalized as a sequence of actions and finite output states. This model has allowed us to investigate the possibility of applying code-based testing methodologies to testing graphical user interfaces. Our findings highlighted challenges and revealed opportunities to adapt code-based testing methodology to verify the correctness of such interfaces. In particular, the "All-OP-DUs" technique provides important error detection capability and can be applied effectively to test GUIs. This paper also introduces Xtester, the GUI testing tool we are currently building to empirically evaluate our proposed testing criteria.

Keywords: Code-based testing, GUI testing, Regression testing, Data-flow testing

1. Introduction

Frameworks and design patterns are typically used in the creation process of complex Graphical User Interfaces (GUIs). Although the supplied code of underlying frameworks is error-free, the assembly process of different *GUI* gadgets can introduce errors and subsequently the end-results can be incorrect. One way to verify the correctness of the end-results is to thoroughly test every *GUI* event and ensure that its results satisfy the specifications. A *GUI* event normally consists of an action sequence. There are many action sequences that one could initiate when interacting with a particular *GUI*. Every possible action sequence can potentially result in a different or similar expected *GUI* state. An expected *GUI* state is a legal state that conforms to specification, given a certain action sequence. An unexpected *GUI* state is an illegal state or *GUI* behavior that may render execution of any further action in a given action sequence useless. Moreover, since action sequences are interlinked and the effect of a given action in an action sequence may depend upon the effects of previous actions, resulting states may not be always reached for verification with the expected legal state. This is mainly due to a possible incorrect *GUI* behavior as a result of one action, which may subsequently prevent further actions in the action sequence from being executed. This implies that after each and every action, during the testing process, the *GUI* state should be examined to determine its legality [1, 2]. This of course requires that the execution process be halted and a thorough examination of the *GUI* state be conducted. Once an illegal *GUI* state or an error is recognized, the execution of a test case or an action sequence must be terminated, and correction steps must be taken.

Recent work in *GUI* testing has focused on building test oracles for *GUIs* [1] and generating test cases [2, 3] while others focused on deriving and adapting suitable code-based coverage criteria [4] or developing new techniques for regression testing [5], cost-effective model-based testing [6] and specification-driven automatic testing [7]. Conventional code-based coverage [8] may not be entirely applicable for *GUI* testing since what matters is not the amount and type of framework code that is exercised, but rather the number of reachable legal states of the *GUI*.

Researchers also worked on capture-and-replay tools [1, 2] to facilitate the *GUI* testing process. These tools capture the user events and *GUI* screens during an interactive session. The recorded sessions are later played back whenever it's necessary to recreate the same *GUI* states.

Other related issues include challenges associated with *GUI* regression testing and those pertaining to inconsistencies in the input/output format across different versions of the software. Possible solutions to answer these challenges are not discussed in this work; however, in future work we intend to address these issues within the *GUI* testing framework presented in this article.

In this paper we introduce a Finite State Machine (FSM)-based graph to represent legal state transitions in a given *GUI*. Based on this graph model, we introduce code-based testing methodologies that best suit the testing process of states in a *GUI*. Our *GUI* state graph model is a variant of the model presented in [3]. We show that our criteria provide an important level of error detection capability, can be used to detect errors effectively, and offer adequate support for testing *GUIs*.

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Section 3 we describe the set of valid objects, properties, and actions in a *GUI,* as well as the different approaches by which these sets can be determined. In Section 4 we introduce our *GUI* state model and subsequently introduce our state-based graph, which we then use to represent control and data-flow information of the *GUI* state model. In this section we also propose our control-flow and data-flow *GUI* testing methodologies. In Section 5 we discuss how state verification can be achieved and explain how the testing verifications considered can be applied to our testing techniques. In Section 6 we informally describe Xtester, the GUI testing tool we are currently building to empirically evaluate our proposed testing criteria. Finally, in Section 7 we wrap-up the paper with pointers to future work and several concluding remarks.

2. Related Work

Most of the work in GUI testing focused on building test oracles for *GUI*s [1, 2], generating test cases [3,4], and automating the testing process [2, 4, 5, 7]. Other work [9] presented a visual Test Development Environment (TDE) for testing *GUI*s. The authors 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. This work was later completed in [1, 2, 4] by creating a testing oracle that would essentially generate an expected state of the *GUI*, pass it to a verifier along with the output state and pass a verdict of whether the states are identical or not.

Other recent approaches on *GUI* testing has been focused on developing new model-driven testing techniques [6, 10], specification-based automatic testing of *GUI*-based Java programs [7], generating test cases for *GUI* using interaction sequences [11], formal methods for design and automatic checking of user interfaces [12], and introspective approaches for marking *GUIs* [13].

The work presented in [1] and [3] was especially valuable to us since it provided an insight into both state modeling and finite state automata of *GUI* specification, respectively. Furthermore, the work in [1] was also very useful to us in adapting the design of Xtester, our proposed *GUI* testing tool. Our prior work on control-flow testing methodologies for visual dataflow languages [14, 15] also provided a useful basis for tackling the work described in this paper.

3. GUIs State Values and Actions

At any given time *t*, a *GUI* can be thought of as having a particular state. A *GUI* state can be represented through its active objects, their values, and the set actions that are allowed on these objects. Next we discuss each category in some detail.

3.1 Objects and Properties

Consider the *GUI* presented in Fig. 1. This *GUI* consists of just one window *w1* and the current set of *w1's* properties and their corresponding values: **window(*w1*); background-color(*w1*, *Grey*);** and **is-current(*w1*)**. Note that the state of *w1*, at any given time *t,* is everything that is currently true for *w1*. Thus, a *GUI* state is related to the type of all available *GUI* objects, as well as all properties related to each one of those objects. An object can have multiple values. Objects' properties may be *fluents* [1]. That is, an object's properties are relations that may evaluate to either true or false, depending on the *GUI* state at time *t*. Some fluents may be undefined in some *GUI* states. For example, if the property of *w2* is requested at time *t*, where *w2* is not an active object at *t*, the value of *w2*'s property would be then undetermined.

There are different approaches to extract objects (windows, labels, buttons, menu-bar, etc.) and their properties. One approach is to manually examine the *GUI* and collect all possible objects, their properties, and actions that can be performed on them [2]. A second approach is to derive the objects and properties directly from the specifications, which include all possible combinations whether they are implicitly or explicitly present in the *GUI* [2]. A third approach, presented in [4], goes one level above the specification of the *GUI* and examines the toolkit used to develop the *GUI* itself. One important observation to note here is that the third approach subsumes the second, and the second subsumes the first. The *subsume r*elationship here is expressed in the completeness of the set of properties an approach covers or provides. For example, the third approach gives the *complete set* of properties of a *GUI* whereas the others give a subset of these properties or what could be called the *reduced set*. In our work we adopt the third approach.

3.2 Actions

Unlike properties, actions are defined to be functions, and have a one-to-one correspondence among states. An action is normally deterministic and leads to one specific *GUI* state. More formally, an action is defined as a function that takes a *GUI* from *CurrentState* to *IntermediateState* resulting in the tuple <*CurrentState, **Action**, IntermediateState*>. An action sequence starts at the *Initialstate* and ends with the *FinalState*. An action sequence example is depicted in Fig. 2a.

Actions have implicit conditions as well. That is, some specific actions cannot be executed to transform the *GUI* from one state to another unless certain conditions are met. For example, the tuple <$State_i$, **set-background-color** (*w1, yellow*), $State_{i+1}$> is not a valid tuple unless the condition **is-open** (*w1*) in $State_i$ evaluates to true. Actions may be extracted based on the specification of a *GUI* [1].

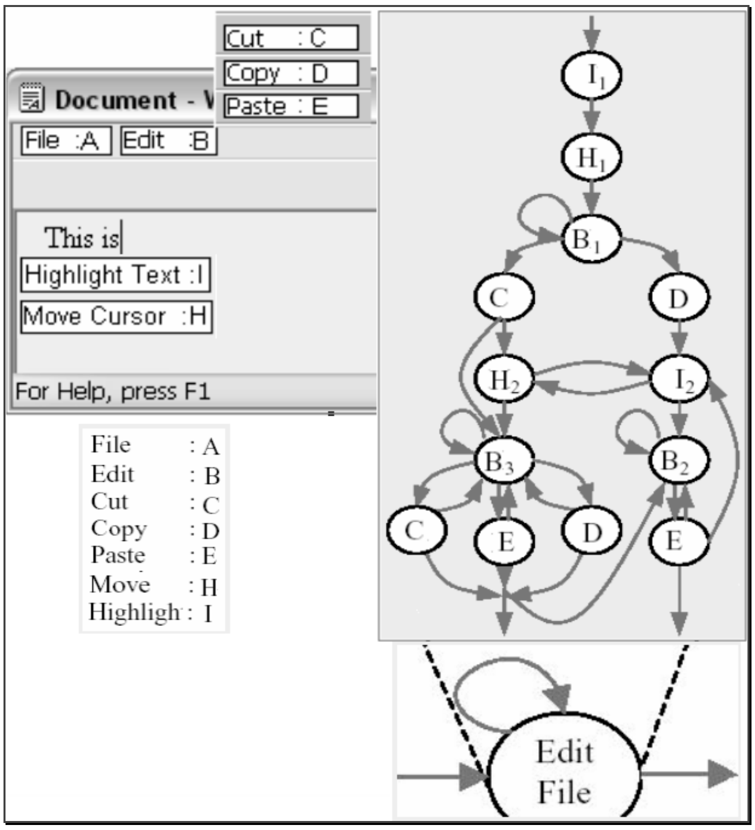

Fig. 1: The *Edit* sub-graph (see also [2] and [3]).

4. GUI State Modeling

Given a software system Φ with a graphical user interface ω, developed in a framework *F*, we model ω as the tuple <*O*, Π, *A*> where, $O = \{o_1, o_2... o_n\}$ is the set of *GUI* Objects $\in$ ω, such that each $o_i \in$ O is a Window, Menu, Button, Text, or any other *GUI* element in $\in$ *F*; $\Pi = \{P_1, P_2... P_n\}$ where each $P_i = \{p_1, p_2... p_n\} \in \Pi$ is the set of properties of each $o_i \in O$, such that each $p_i \in P_I$ is any property such as background color, font, is-open, or any other property provided to $o_i \in F$; and $A = \{a_1, a_2... a_n\}$ is a set of actions that can be applied to $p_i \in P_I$ of o_i.

4.1 A State-based Graph for GUIs

To utilize the *GUI* state model in a way that allows us to apply code-based testing, we construct for a given ω a state graph $SG_{(GUI)}$ (ω, S_e), such that: each object $o_i \in O \in$ ω is a node in $SG_{(GUI)}$, with the sets of its properties and allowed legal actions attached to it; $S_e = \{ s_{e(1)}, s_{e(2)}, ..., s_{e(n)}\}$ is the set of edges $\in SG_{(GUI)}$, where an edge $s_{e(i)}$ is constructed from an object o_i to another object o_j iff there is a legal action $a_i \in A$ that can be applied to the property $p_i \in P_i$ of o_i, such that a_i takes ω from *CurrentState* to *IntermediateState* resulting in the tuple <*CurrentState*, $\boldsymbol{a_i}$, *IntermediateState*>.

The $SG_{(GUI)}$ is therefore a deterministic finite state automata (FSA). The latter has been extensively used to model sequential and combinatorial systems, compiler construction, and *GUI* specifications and testing [11, 16]. In our work, we construct $SG_{(GUI)}$ by adopting techniques that are based on the use of formal specifications for construction *GUI*s [3, 16].

To informally illustrate the construction process of a $SG_{(GUI)}$, consider the *GUI* example that is depicted in Fig. 1, and the sub-graph (Edit menu) of its corresponding $SG_{(GUI)}$. The *GUI* of Fig. 1 depicts a simplified version of WordPad©. Nodes in the sub-graph represent every object $o \in O$ of the *GUI* sub-graph. Nodes in the sub-graph having subscripts are there to facilitate the discussion and illustration. They are essentially aliases to the same object to which the capital letter points to. Edges in the sub-graph represent all possible single action sequences. For example, the edge (*I1*, *H1*) is constructed, based on the *GUI* specification [3, 16], and represents a highlight action, followed by cursor move action. Also an edge can be constructed from a node to itself. For example, edge (*B1*, *B1*) represent two consecutive clicking actions on the *GUI* object menu Edit.

4.2 Test Adequacy Criteria for GUIs

In this section we describe the process of adapting control-flow and data-flow code-based testing techniques [8] to test *GUI*s.

4.3 Control-flow Testing Methodology

***Definition* 4.1.1:** Given a *GUI* ω with its state graph $SG_{(GUI)}$, a test (action sequence) a_s exercises a node n in $SG_{(GUI)}$ if a_s causes the traversal of a path p through $SG_{(GUI)}$ that includes n.

Analogous to statement coverage [8], a test suite, a set of action sequences $A_{s,}$ is node-adequate for ω iff for each dynamically executable node n in $SG_{(GUI)}$ there is at least one test in A_s that exercises n. This defines the all-nodes criteria, and requires that all possible states in a *GUI* be covered. This criterion, while weak, ensures that every state can be reached via any edge. Moreover, once a state is reached, any illegal action can be detected since these nodes "pack" with them all possible actions.

Definition* 4.1.2:** A test a_s exercises an edge ($s_{e(i)}$, $s_{e(j)}$) in $SG_{(GUI)}$ if it causes the traversal of a path in $SG_{(GUI)}$ that includes ($s_{e(i)}$, $s_{e(j)}$). A test suite A_s is edge-adequate for ω iff for each edge ($s_{e(i)}$, $s_{e(j)}$) $\in SG_{(GUI)}$, there is at least one test in A_s that exercises ($s_{e(i)}$, $s_{e(j)}$). This is analogous to the *branch coverage* in imperative languages [8]. This criterion is slightly stronger that the node adequacy criteria described in ***Definition 4.1.1. It ensures that every state can be reached via every possible edge or action.

4.4 Dataflow Testing Methodology

At any time t, a GUI state is described in terms of the specific *objects* it currently contains, and the current values of their properties. Properties of GUIs can assume different values depending on the object it is referring to. Furthermore, a property is *fluent*; and can be undefined at certain times. In other words, a property is a recipient for a certain value, at a certain point in time t. Thus, the "life" of a property in a GUI, is analogous to the "life" of a variable in imperative languages, where a variable is first undefined upon declaration, then *set* during the program execution, and eventually *used*. With imperative languages, the procedure or the scope defines the accessibility of a variable. With GUI however, a GUI object such as Window, can be referred to as a *scope*, or a *procedure* where some properties are *defined*, or *set* (used). This approach of modeling GUI behavior is the basis of our approach in adapting code-based data-flow testing techniques for testing GUIs. More formally, and as a summary of the above observations, we define the following elements:

– *Objects*:	Scopes
– *Properties*:	Variables
– *Property-values*:	Values that a variable can take
– *Actions*:	Setting the *properties (values)* of an *object*
– *Creating/killing objects*:	Similar to the notion of scopes

***Definition* 4.1.3**: an object property or variable denoted O_P is said to be *live,* if there is a path in $SG_{(GUI)}$ from the node in which it was first defined to where it is referenced, such that the *GUI* object is not killed or no longer available in ω.

***Definition* 4.1.4**: an O_P is said to be defined or set, when the scope (*object*) is first invoked, or when there exists an action in the action sequence that accesses and changes the value of an O_P; that is if the object is already invoked. Invoking an object in this context would include actions such as clicking the Edit menu or opening a new window. In the first case, the scope is the object "Edit menu". In the second case, the scope is the object "Window".

***Definition* 4.1.5**: an O_P is said to be used, when there exists an action in the action sequence that implicitly inquires about the value of this property. Inquiring about the value of an O_P is often implicit in an action. For example given the action sequence: <$State_i$, **set-background-color** (*w1*, *yellow*), $State_{i+1}$>, this action would implicitly inquire whether the property **is-open**(*w1*) evaluates to *true*. Therefore, such inquiries are regarded as a use of the *w1*'s current property. It is important to note that we make no distinction in our definition between a computational use or *c-use* and predicate use or *p-use* [8]. This is mainly due to it's irrelevancy in a state graph, since values of properties are consulted for transitions.

***Definition* 4.1.6**: All-O_P-Uses criterion. A set *P* of execution paths $\in SG_{(GUI)}$ is said to satisfy this criterion iff for any use of an O_P, there exists at least one action in an action sequence (input) that causes the traversal of at least one path which is feasible, and on which a definition reaches this use.

An infeasible path here is a path that is based on faulty events. This is known as the Oracle problem [1]. Infeasible paths can be determined by making sure that only appropriate edges (paths) are traversed in the $SG_{(GUI)}$. Any sub-path that contains edges traversed incorrectly is considered to be infeasible. An alternate solution for determining infeasible paths is provided in [3].

This criterion requires that all uses of O_P are exercised by testing, and that each and every one of them falls on at least one path in which there is a definition of this variable.

This criterion catches errors where there is a use of a property of an object when the object is not defined. To illustrate this issue consider a simple execution of a *GUI* :

Initialstate
Open-window(w_1)
Set-background–color(w_1, Blue)
Output state1;
Resize-window(w_2, 20, 20)
Output state2;

Here errors like referencing a certain object's properties when the object was not defined could be caught. That is, a reference to *w2* is considered to be a reference or a use without having defined the object that owns it. On the other hand, consider the *GUI* example depicted in Fig. 2 (a, b and c): Fig. 2a shows the initial *GUI* state and the action sequence to be executed; Fig. 2b shows the correct output; and Fig. 2c shows the resulting *GUI* state after the execution of the specified action. As depicted in Fig. 2c, the Help menu has disappeared. This is an incorrect resulting behavior of the *GUI* under the given action sequence. The All-O_P-Uses criterion would fail to catch such an error. Thus, a more rigorous criterion is required. We describe this next.

***Definition* 4.1.7**: All-O_P-Dus (definition and uses) criterion. A set P of execution paths $\in SG_{(GUI)}$ is said to satisfy this criterion iff for all definitions an O_P, and all paths q through which this definition reaches a use of the object property O_P, there exist at least one path p in P which is feasible and such that q is a sub path of p on which a definition reaches this use. Consider again the erroneous example of Figure 2. With the All-O_P-Dus, such an error is caught since the path does not match any of the static O_P Dus chains.

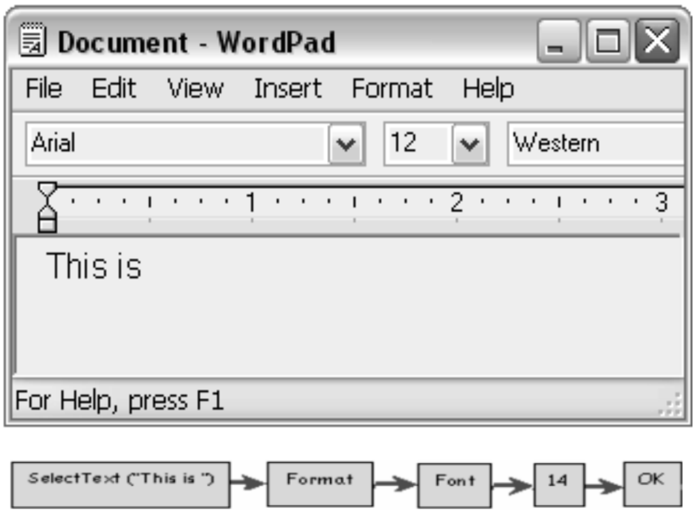

Fig. 2a: An initial *GUI* state and an action sequence to be executed

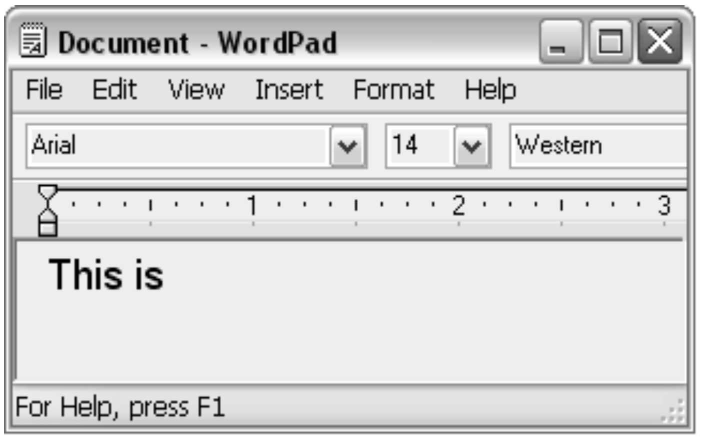

Fig. 2b: Correct *GUI* behavior.

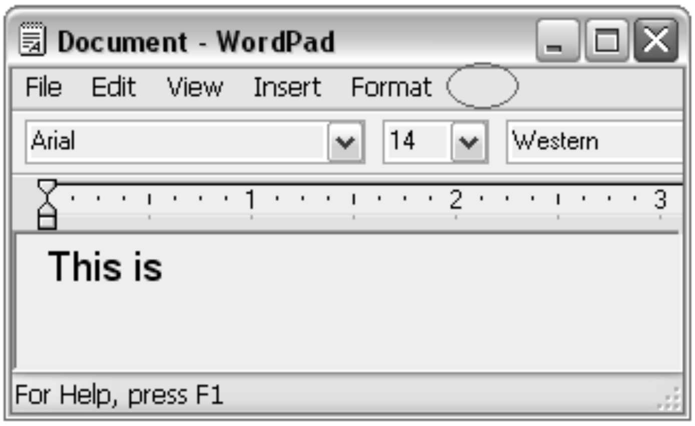

Fig. 2c: Incorrect *GUI* behavior

5. State Verification

To determine whether a specific action has lead to a correct behavior, one should not only examine the output of the *GUI* (screen shot [1]), but also the state of the *GUI* as well. Consider again the *GUI* example presented in Figure 2. Examining only the output of the action sequence could render the error undetected since the font of text "This is" in the *GUI* was changed to the desired size; however, by examining the *GUI* state as well, such an error would be detected. Thus, in our work we not only examine the output but also the *GUI* states as well. To do so, we adapt three levels of testing/verification [1] which will be discussed next.

5.1 Levels of Verifications

We define three levels of state verification: Modified Properties Verification; Relevant Properties Verification; and Complete Properties Verification. Each level will be explained next.

1 – *Modified Properties Verification*. At this level, as described in [1], verification via comparison is made for only those properties that are expected to change. Although considered efficient, this level fails to report errors that involve changing the values of properties that were not expected to change. This was illustrated in the example depicted in Fig. 2.

2 – *Relevant Properties Verification*. This level of testing was also described first by [1]. Here, all the properties in the *reduced set* are examined. This level is more extensive than the modified-properties described above; however, it fails to detect errors that change the value of properties not present in the *GUI* specification. For example, changing the background color in the *GUI* of the WordPad© is not part of the specification; however doing so is a part of the toolkit options in which the *GUI* was developed).

3 – *Complete Properties Verification*. This is the most extensive level of testing. Here, a check is made to all the properties used by a toolkit or language to develop the *GUI* provides [1]. It is important to note that testers can choose a hybrid approach by combining these testing levels. For example, one could choose to apply the modified-properties verification after every step and the reduced-properties verification after every n^{th} step.

For every testing technique that we defined in ***Definitions 4.1.1*** to ***Definitions 4.1.6***, these verification levels can be used. For example, in the All-O_p-Dus, we can use any verification level. Using the Complete Properties Verification level would provide a way to check every state, and thus give a complete and exhaustive testing – however, it may not be very practical. Using the Relevant Properties Verification level, errors such as the one described in Fig. 2 would not be detected.

6. The Xtester

This section provides a summary description of the Xtester, our proposed GUI testing tool, depicted in Fig. 3. It is intended to provide the reader with a way to visualize the eventual testing process and use of the testing criteria we have proposed in this work. The Xtester is designed to be essentially composed of four basic parts: Glex, Monitor, VirtualX, and Verifier. Two of Xtester's main parts, the Monitor and the Verifier, will be adapted from [1] and modified according to the current and future needs of our approach. More details on Xtester's main components are provided next.

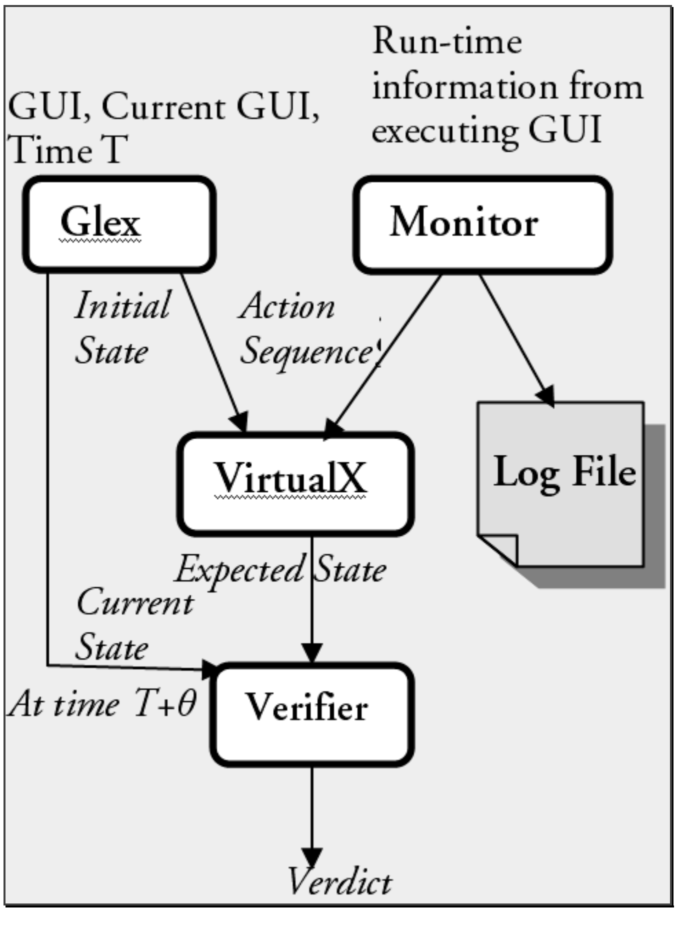

Fig. 3: Overview of the Xtester testing tool

The Glex: is essentially a graphical analyzer that accepts as input a GUI model in the format described in Section 4, a GUI screenshot, and a time *t* to mark the starting of the analysis of the GUI. Based on the input, Glex then produces a *GUIState*. Glex has the following function prototype: *GUIState* Glex*(Model GUIModel, Time t).*

The Monitor: as the name indicates, it monitors the user actions (mouse clicks, resizing, moving objects, etc.), and transforms them into action sequences, as specified in Section 3. The transformed action sequences are later outputted to VirtualX. The Monitor also produces a Log file that contains the set of actions invoked by the user and the sate of the GUI ω after the execution of these actions. The Monitor has the following function prototype: *ActionSequence* Monitor*(Run-time information, FILE * logfile, Time timeduration).*

The Log file: this is generated by the Monitor, and will have the the following grammar:

Logfile → *InitialGUIState*; "\n" *Block*
Block → *Action OutputGUIState* "\n" *Block*

The Log file is essentially a file that keeps track of the action that users perform and the resulting *GUI* state ω The Log file initially starts with the *InitialGUIstate*. At the end of the monitoring session (the duration of the monitoring session could be set by the user as an argument to the Monitor), the Log file will be later analyzed to extract all instances where properties are either *live, set,* or *killed.*

The VirtualX: This part is essentially a virtual executer that generates the *expected-state* of a GUI given an action (or a sequence of actions), and the current state of the GUI. The function prototype of VirtualX can be seen as: *GUIState* VirtualX*(Action a1, GUIState CurrentState).*

The Verifier: This part will pass the verdict as to whether these two states are identical, based on the level of testing specified. The function prototype of the Verifier can be seen as: *Boolean* VirtualX*(State S1, State S2, int Leveloftesting).*

7. Conclusion and Future Work

In this paper we have identified and described challenges related to testing *GUIs* and outlined strategies for tackling them. We have presented a *GUI* model that represents *GUI* actions in terms of their preconditions and effects. Our model represents the execution of a *GUI* as a sequence of actions and output states. This model was then incorporated into a state graph that made it possible to apply code-based testing methodologies to test *GUIs*. The All-O_P-Dus criterion, in particular was useful, and can provide important error detection ability. We have also introduced the planned design of Xtester, the GUI testing tool we are currently developing to empirically evaluate the usefulness of our proposed testing criterion.

In addition to fully developing and thoroughly exercising Xtester as well as further evolving the formalism presented in this paper, planned future work includes performing regression testing on *GUI*s as well testing *GUI*s that change over time.

References

[1] A.M. Memon, M.E. Pollack and M.L. Soffa. Automated Test Oracles for GUIs, in *Procs. of the ACM SIGSOFT 8th Intl. Symp. on the Foundations of Software Engineering (FSE-00)*, D.S. Rosenblum, editor, ACM Press, 2000, pp. 30-39.
[2] A.M. Memon, M.E. Pollack and M.L. Soffa. Using a Goal-driven Approach to Generate Test Cases for GUIs, in *Procs. of the 21st Intl. Conf. on Software Engineering (ICSE'99)*, ACM Press, 1999, pp. 257-266.

[3] B. Fevzi. Finite-State Testing and Analysis of Graphical User Interfaces, in *Procs. of the 12th Intl. Symp. on Software Reliability Engineering (ISSRE-2001)*, IEEE Computer Press, 2001, pp. 34-43.

[4] A.M. Memon, M.E. Pollack and M.L. Soffa. Coverage Criteria for GUI Testing, in *Procs. of the 8th European Software Engineering Conf.,* held jointly with the 9th ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering (ESEC/FSE-9)*,* 2001, pp. 256-267.

[5] A.M. Memon and M.L. Soffa. Regression Testing of GUI, in *Procs. of the 9th European Software Engineering Conf.,* held jointly with the 11th ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering (ESEC/FSE'03)*,* 2003, pp. 118-127.

[6] Q. Xie. Developing Cost-Effective Model-Based Techniques for GUI Testing, in *Procs. of the 28th Intl. Conf. on Software Engineering (ICSE'06)*, ACM Press, 2006, pp. 997-1000.

[7] Y. Sun and E.L. Jones, Specification-Driven Automatic Testing of GUI-Based Java Programs, in *Procs. of the 42nd ACM Southeast Conf. (ACMSE'04)*, 2004, pp. 140-145.

[8] P. Frankl and E. Weyuker. An Applicable Family of Data Flow Criteria, *IEEE Trans. on Software Engineering,* 14 (10), 1988, pp. 1483-1498.

[9] T. Ostrand, A. Anodide, H. Foster and T. Goradia. A Visual Test Development Environment for GUI Systems, in *ACM SIGSOFT Software Engineering Notes*, 23 (2), *Procs. of the ACM SIGSOFT Intl. Symp. on Software Testing and Analysis*, 1998.

[10] M.Viera, J. Leduc, B. Hasling, R. Subramanyan and J. Kazmeier. Automation of GUI Testing Using a Model-driven Approach, in *Procs. of the 1st ACM Workshop on Automation of Software Test (AST'06)*, 2006, pp. 9-14.

[11] L. White and H. Almezen, Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences, in in *Procs. of the 11th Intl. Symp. on Software Reliability Engineering (ISSRE-2000)*, IEEE Computer Press, 20010, pp. 110-119.

[12] J. Berstel, S. Crespi Reghizzi, G. Russell and P. San Pietro. A Scalable Formal Method for Design and Automatic Checking of User Interfaces, in *Procs. of the 23st Intl. Conf. on Software Engineering (ICSE'01)*, ACM Press, 2001.

[13] G.R. Gray and C.A. Higgins. An Introspective Approach to Marking Graphical User Interfaces, in *Procs. of the 11th Annual Conf. on Innovation Technology in Computer Science Education (ITiCSE'06)*, 2006, pp. 43-47.

[14] M.K. Karam and T.J. Smedley. A Control-Flow Testing Methodology for Visual Dataflow Languages, in *the IEEE Symposium on Human Centric Computing*, Italy, 2001.

[15] M.K. Karam, Using Visual Augmentations to Test Control Interactions in Visual Dataflow Languages, in *Procs. of the 5th IEEE Intl. Symp. on Signal Processing and Information Technology (ISSPIT'05)*, 2005, pp. 639-645.

[16] D.L. Parnas. On the Use of Transition Diagrams in the Design of User Interface for an Interactive Computer System, in *Procs. of the 24th ACM National Conf.*, 1969, pp. 379-385.