JCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-3, Issue-1 E-ISSN: 2347-2693

# Testing Event Driven Systems By Using Observe-Model-Exercise Paradigm With Unknown Input Spaces

Deepan R* and M.S.Geetha Devasena

*Department Of CSE , Sri Ramakrishna Engineering College, Coimbatore-22,India*
**www.ijcaonline.org**

Received: Dec/26/2014 Revised: Jan/8/2015 Accepted: Jan/20/2015 Published: Jan/31/2015

***Abstract*—** In software engineering, graphical user interface testing is the process of testing a product's graphical user interface to ensure that it meets the written specifications. This is normally done through the use of a variety of test cases. To generate a set of test cases, test designers attempt to cover all the functionality of the system and fully exercise the GUI itself. The difficulty in accomplishing this task is twofold: to deal with domain size and with sequences. In addition, the test faces are more difficult in case of regression testing. In this work, we develop a new paradigm for GUI testing, one that we call Observe-Model-Exercise (OME) to tackle the challenges of testing context-sensitive GUIs with undetermined input spaces. Starting with an incomplete model of the GUI's input space, a set of coverage elements to test, and test cases, OME iteratively observes the existence of new events during execution of the test cases, expands the model of the GUI's input space, computes new coverage elements, and obtains new test cases to exercise the new elements. The experimental results proves that the proposed work is better than the previously existing works.

***Keywords-*** OME;User Interface;Context-sensitive GUIs;Test Case Generation;Quality Concepts

## I. INTRODUCTION

Event-driven architecture (EDA) is a software architecture pattern promoting the production, detection, consumption of, and reaction to events. An event can be defined as "a significant change in state".A graphical user interface (GUI) is a human-computer interface (i.e., a way for humans to interact with computers) that uses windows, icons and menus and which can be manipulated by a mouse (and often to a limited extent by a keyboard as well). GUIs stand in sharp contrast to command line interfaces (CLIs), which use only text and are accessed solely by a keyboard. The most familiar example of a CLI to many people is MS-DOS. Another example is Linux when it is used in console mode.

System testing of software applications with a graphical-user interface (GUI) front-end requires that sequences of GUI events that sample the application's input space, be generated and executed as test cases on the GUI. However, the context-sensitive behaviour of the GUI of most of today's non-trivial software applications makes it practically impossible to fully determine the software's input space. The tester does not have a complete picture of the GUI's input space, i.e., the set of all possible sequences of user interface events. The tester is never supplied a blueprint of the GUI or its set of allowable workflows. But there is no way for a tester to determine which sequences are missed and which should not be allowed.

## II. LITERATURE SURVEY

A GUI Based Visualization Tool for Sequence Networks - David C. Yu, Member Haijun Liu, Student Member Fengjun Wu – 1998.[4]

The fault analysis is an important part of the power system undergraduate curriculum. There are two main fault analysis are present. Those are symmetrical fault and asymmetrical faults. A symmetric or balanced fault affects each of the three phases equally. In transmission line faults, roughly 5% are symmetric. This is in contrast to an asymmetrical fault, where the three phases are not affected equally. An asymmetric or unbalanced fault does not affect each of the three phases equally. In this work a Windows based Graphical User Interface (GUI) software tool is evaluated to facilitate the teaching and learning of sequence networks. This software is written in Microsoft Visual Basic. The software provides a friendly and easy to use tool to aid the students in better visualizing the effects of the sequence diagram and sequence current in the fault study.

Hierarchical GUI Test Case generation using automated planning - Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa – 2001.[5]

Graphical user interfaces (guis) have become an important and accepted way of interacting with today's software. Although they make software easy to use from a user's perspective, they complicate the software development process. The widespread use of GUIs for interacting with software is leading to the construction of more and more complex GUIs. Testing GUIs is more complex than testing conventional software, for not only does the underlying software have to be tested but the GUI itself must be exercised and tested to check whether it confirms to the GUI's specifications. Even when tools are used to generate GUIs automatically, these tools themselves may contain errors that may manifest themselves in the generated GUI leading to software failures. Hence, testing of GUIs

© 2015, IJCSE All Rights Reserved 

continues to remain an important aspect of software testing.

In this work a new technique to automatically generate test cases for GUIs that exploits planning, a well-developed and used technique in artificial intelligence is proposed. Given a set of operators, an initial state, and a goal state, a planner produces a sequence of the operators that will transform the initial state to the goal state. Our test case generation technique enables efficient application of planning by first creating a hierarchical model of a GUI based on its structure. In developing a planning system for testing GUIs, the first step is to construct an operator set for the planning problem. And then the test designer begins the generation of particular test cases by identifying a task, consisting of initial and goal states.

Studying the Fault-Detection effectiveness of GUI test cases for rapidly evolving software - Atif M. Memon and Qing Xie – 2005.[7]

Many of the software applications are developed and maintained by multiple programmers, often geographically distributed, who work on parts of the overall application code. While leading to improved code churn rates, this practice also leads to problems. For example, developers may not realize that they have inadvertently broken parts of the code. Consequently, rapid-feedback-based quality assurance mechanisms are integrated into the development and maintenance cycle. In this work a major weakness of current smoke regression testing techniques, i.e., their inability to automatically (re)test graphical user interfaces (GUIs) are analysed. This work builds upon several aspects of automated GUI testing. Those are Size of a smoke test suite, complexity of test suits, Characteristics of test suits. Several contributions are made to the area of GUI smoke testing. First, the requirements for GUI smoke testing are identified and a GUI smoke test is formally defined as a specialized sequence of events. Second, a GUI smoke regression testing process called Daily Automated Regression Tester (DART) that automates GUI smoke testing is presented. Third, the interplay between several characteristics of GUI smoke test suites including their size, fault detection ability, and test oracles is empirically studied.

The main design goal of DART is to automate GUI smoke testing. A test designer uses a process called the “DART process” to realize this automation. The six modules developed in the DART operations are: The developer (or test designer) identifies the AUT, DART analyzes the (baseline) AUT’s GUI structure, DART computes the total number of possible smoke test cases, DART’s automated test case generator, A test oracle generator automatically creates for each test case, The development team uses change requests and bug reports to modify the AUT, the operating system’s task scheduler launches DART, Test cases are executed (using a test case executor) on the instrumented modified AUT, the developers examine the reports.

Test-Driven GUI Development with testing and abbot - Alex Ruiz and Yvonne Wang Price – 2007[6]

A graphical user interface (GUI) is a human-computer interface (i.e., a way for humans to interact with computers) that uses windows, icons and menus and which can be manipulated by a mouse (and often to a limited extent by a keyboard as well). Testing GUIs can make the entire system safer and more robust. Any GUI, even one providing only the simplest capabilities, likely encloses some level of complexity. The more user-friendly a GUI is, the more complexity it might be hiding from the user. Any complexity in software must be tested because code without tests is a potential source of bugs. A well-tested application has a greater chance of success.

GUIs are complex pieces of software. Testing their correctness is challenging for several reasons: Those are Tests must be automated, Conventional unit testing, involving tests of isolated classes, is unsuitable for GUI components, GUIs respond to user-generated events, The room for potential interactions with a GUI is huge.

Abbot (http://abbot.sourceforge.net) is a Java library for testing Swing GUIs that supports both the record/playback and programmatic GUI testing styles. JUnit introduced automated unit tests to Java developers. Although it does this successfully, JUnit aims to test classes in isolation, leaving developers without the extra features and flexibility necessary for higher levels of testing.

Developing a single model and test prioritization strategies for event-driven software - Rene´e C. Bryce, Sreedevi Sampath and Atif M. Memon – 2011.[8]

Event-Driven software (EDS) is a class of software that is quickly becoming ubiquitous. All EDSs take sequences of events (e.g., messages and mouse-clicks) as input, change their state, and produce an output (e.g., events, system calls, and text messages). Examples include Web applications, graphical user interfaces, network protocols, device drivers, and embedded software. These EDSs pose a challenge to testing because there are a large number of possible event sequences that users can invoke through a user interface.

A GUI is the front-end to a software’s underlying back-end code. An end user interacts with the software via events; the software responds by changing its state, which is usually reflected by changes to the GUI’s widgets. The complexity of back-end code dictates the complexity of the front-end. Due to their user-centric nature, GUI and Web systems routinely undergo changes as part of their maintenance process. New versions of the applications are often created as a result of bug fixes or requirements modification.

 © 2015, IJCSE All Rights Reserved 

In this work the model to define generic prioritization criteria that are applicable to both GUI and Web applications is proposed. The ultimate goal is to evolve the model and use it to develop a unified theory of how all EDS should be tested.

## III. PROPOSED SYSTEM

This project proposes a new paradigm for GUI testing, one that we call Observe-Model-Exercise* (OME*) to tackle the challenges of testing context-sensitive GUIs with undetermined input spaces. Starting with an incomplete model of the GUI's input space, a set of coverage elements to test, and test cases, OME* iteratively observes the existence of new events during execution of the test cases, expands the model of the GUI's input space, computes new coverage elements, and obtains new test cases to exercise the new elements.

To provide focus, It only consider the behaviors directly caused by the order of GUI events. Other potential causes of "context-sensitivity" such as timing and multiple-user profiles are left for future work. Specifically, we develop a new paradigm for GUI testing, one that we call Observe-Model-Exercise (OME). The key feature of OME_ is its opportunistic use of test execution for model enhancement. More specifically, we now observe the existence of new events either during Ripping or test execution, create or enhance our EFG+ model an extension of our EFG model, and exercise the newly observed GUI events in test cases using test adequacy criteria. As new test cases are generated and executed.

### A. *Creating Event Flow Graph*

Event flow graph is created with the help of Ripper. The functionality of ripper is to traverses towards the GUI events and returns the behaviour of those events. Ripper is not used to testing the functionality behaviours of GUI. Instead of that, GUI is used to computes follows relationship by opening and closing the as many windows as possible. The follows relationship is created as like follows event ex follows event ey which means that event x may be executed immediately after the event y. This follows relationship is represented as edge between two events. That edge is used to denote the follows relationship among the nodes. Ripper is based on depth first search (DFS) traversal. That is it will start from the main window and will extract all widgets required to prove the events. Based on the follows relationship that is extracted while traversing towardsthe nodes, event flow graph will be created. The EFG will define the relationship among the every possible events present in GUI by using the follows relation. EFG is nothing but the GUI blueprint to the users which will enable the users to flexible and efficient access on the systems with GUI.

*Algorithm: Construct Mapping*

Input: <(e1, α (S1)), …., (en, α (Sn))>: Executed Sequence
Input: CM: Context- Aware Mapping
Input: α (I): Events enabled in initial state

```
1.    T = Ø
2.    For i = 1 □ n do
a.    For all ej ϵ α (Si) do
i.    T.addEdge (ei, ej)
b.    End For
3.    End For
4.    ME □ getModelElements (T)
5.    For all me ϵ ME do
a.    If firstEvent (me) ϵ α (I) then
i.    contextSeq = NONE
b.    else
i.    contextSeq = searchPath (me, T)
c.    end if
d.    truncate (contextSeq)
e.    if me ϵ/ CM then
i.    CM.addEntry(me, contextSeq)
f.    Else
i.    contextSeqold □ lookup (CM, me)
ii.   if |contextSeqold| > |contextSeq| then
1.    CM.updateEntry (me, contextSeq)
iii.  End If
g.    End if
6.    End for
7.    Return CM: Updated Context-aware mapping
```

### B. *Generating the test cases*

In this module, the test case generation is done for evaluating the possible faults that can be occur in the system. The test case generation is used to exercise the new events possible in the system.

The number of faults can be identified while transferring among the events by using the test cases generated.The possible test cases which are executable will be generated from the available EFG diagram. While creating EFG through ripper, some of the hidden events may be missed. For example the events which are in disabled mode cannot be considered by the ripper mechanism.

Ripper mechanism will only evaluate the events which are all enabled. In this module, the possible test cases that can be executed are generated. The test cases are generated by finding out all the possible edges that are available from the initial nodes. In this module all the test cases are generated from the all available EFG edges.

*Algorithm: Collecting new widget states*

```
Method [] methods = widget.getClass (). getMethods();
For (Method m : methods)
{
        String methodname = m.getName ();
        If (methodName.startsWith ("get"))
{
        Property = methodName. substring (3);
Value = m.invoke (widget, new Object [0]);
}
If (methodName.startsWith ("is"))
{
```


© 2015, IJCSE All Rights Reserved

```
        Property = methodName.subString (2);
Value = m.invoke (widget, new object [0])
}
```

### C. *Executing the test cases*

In this module, the newly generate test cases will be executed simulataneously on the same application in order to derieve an unlocalized test cases. To identify the already evaluated test cases unique signature concept is introduced. It is based on using a combination of certain parts of the state of the widget and its container (e.g., window). It cannot use the entire state for identification because it will contain some property values that change during the GUI's execution but do not play any role in identifying that widget. For example, the value of the text property for a JTextField object will change when the text changes; the enabled property changes when the object is enabled/disabled. Such properties cannot be used for our signature because any change to their values will indicate a new widget, which would be incorrect. We are, in some sense, defining equivalent states of widgets by using a subset of properties to uniquely identify widgets. More formally, we define the signature, Csig, for a container C as follows:

Cstate □ <(p1,v1), (p2,v2), …, (pn,vn)>,

<vi, … vk> □ select (filterp, Cstate),

Csig □ Φ (Øi (vi), …., Øk (vk))

Where the user defines, per GUI, filterp, a specification of a subset of the container's properties and transformations Øi . . . Øk on the values of the properties. The function select returns the values of the properties specified by filter p and function F is a hash function on the transformed values. Along similar lines, we define the signature, wsig, for a widget w in a container with signature Csig, as follows:

Wstate □ <(p1,v1), (p2,v2), …, (pn,vn)>,

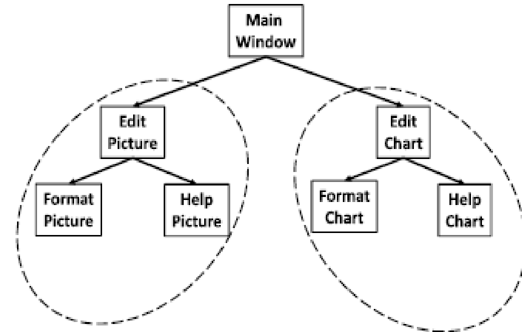
<vi, … vk> □ select (filterp, wstate),

wsig □ r (Csig, γi (vi), …., γk (vk))

where filterp and γi, … γk are user defined; and function r is a hash function on the transformed values and the container's signature.

### D. *Enhance EFG with context aware mapping*

In this module techniques to incrementally enhance the EFG is introduced. To explain these steps, we revisit two important terms in GUIs: modal and modeless windows. At any time during GUI interaction, a user is allowed to execute events within a modal window and any modeless window that was opened from the modal window. At no time can the user jump between modal windows without explicitly terminating them. Moreover, the user cannot interleave events that belong to modeless windows associated with different modal windows. Again, the user must explicitly terminate the modal window that is associated with the modeless window, explicitly invoke the other modal window, open the modeless window, and invoke any of its constituent events. A part of MS Word's window hierarchy is shown in below Fig.




Edit Picture and Edit Chart are modal windows whereas Format Picture, Help Picture, Manage Template, and Help Chart are modeless. Consider events x, y, z, a, b, and c. A user may execute x, y, and z together because they are all contained in Edit Picture's window group; similarly, events a, b, and c may be executed together. However, these two sets of events cannot interleave without their modal windows being explicitly invoked and terminated. The above behavior of GUI windows to restrict sets of events leads to the definition of a new term that we call the scope of an event. We define the scope of an event e as the set of events contained in the group of modal and modeless windows to which e belongs. We use scope in an algorithm to incrementally and efficiently enhance the EFG model.

**Algorithm:** *Enhance EFG Model*

Input: (N, E): EFG
Input: e: event executed

1. AE □ getAllEventsAfter (e)
2. For all ei ϵ AE do
a. If ei ϵ/ N then
i. N.addNode (ei)
b. End if
c. If (e, ei) ϵ/ E then
i. E.addEdge (e, ei)
d. End if
e. Scopei □ getScope (ei)
f. For all eij ϵscopei do
i. If not (structural (ei,j)) then
1. If (eij, ei) ϵ/ E then
a. E.addEdge (eij, ei)
2. End if
ii. End if
g. End for
h. End for
i. Return (N, E): updated EFG

© 2015, IJCSE All Rights Reserved 

### *E. Performance Evaluation*

The performance is done to compare our proposed methodology with the existing techniques. The improvement and efficiency of our proposed methodology is proved to be better than the already existing methodologies. The accuracy and time complexity of our methodology is improved by comparing with the already proposed methodologies.

## IV. RESULTS AND DISCUSSION

The performance evaluation of our proposed work is done by comparing the proposed work with the existing approaches based on the time taken and performance of the test cases for finding out the more number of faults.

### *A. Time Complexity*

The amount of time taken by test cases to finding faults for both existing work and proposed work are compared as follows. The graph shows that our proposed work consumes less time than the existing work for finding out the faults.

### *B. Test case performance*

The performance of test cases is evaluated by finding the faults it can be detect. The test case performance increases when it can find out the more number of faults. The test case performance for both existing and proposed work is compared in the following graph which shows that proposed work is better than the existing work.

## V. CONCLUSION

System testing of software applications with a graphical-user interface (GUI) front-end requires that sequences of GUI events, that sample the application's input space, be generated and executed as test cases on the GUI. GUI testers routinely miss allowable event sequences, any of which may cause failures once the software is fielded. And the tester may fail to discover that the softwares implementation allows the execution of some disallowed sequences.

In our work Observe model exercise mechanism is proposed. Our approach used the GUI information to extract key identifiers for the parameterized widgets (i.e., widgets that accept input values such as textboxes) in the GUI and found appropriate valid and invalid test data using an online search. Our preliminary experiments with five GUI-based applications showed that the proposed technique is feasible and applicable.

## REFERENCES


[1] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, "Bringing White-Box Testing to Service Oriented Architectures through a Service Oriented Approach," J. Systems and Software, vol. 84,pp. 655-668, Apr. 2011.

[2] N.R. Krishnaswami and N. Benton, "A Semantic Model for Graphical User Interfaces," Proc. 16th ACM SIGPLAN Int'l Conf.Functional Programming (ICFP '11), pp. 45-57, 2011.

[3] T. Pajunen, T. Takala, and M. Katara, "Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework,"Proc. IEEE Fourth Int'l Conf. Software Testing, Verification and Validation Workshops (ICSTW), pp. 242-251, Mar. 2011.

[4] David C. Yu, Member Haijun Liu, Student Member Fengjun Wu, "A GUI Based Visualization Tool for Sequence Networks", IEEE Transactions on Power Systems, Vol-13, No-1,pp.247-263,February 1998.

[5] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa, "Hierarchical GUI Test Case generation using automated planning", IEEE transactions on software engineering, vol-27, no-2,pp.453-464, Feb 2001.

[6] Alex Ruiz and Yvonne Wang Price, "Test-Driven GUI Development with testing and abbot", IEEE transactions on software engineering, vol-28, Issue-4,pp.117-134, Jun 2003.

[7] Atif M. Memon and Qing Xie, "Studying the Fault-Detection effectiveness of GUI test cases for rapidly evolving software", IEEE transactions on software engineering, vol-31,Issue-10,pp.944-952,October 2005.

[8] Rene´e C. Bryce, Sreedevi Sampath and Atif M. Memon, "Developing a single model and test prioritization strategies for event-driven software", IEEE transactions on software engineering, vol-37, no-1,pp.812-830,Jan 2011.

[9] Izzat Mahmoud Alsmadi, "Using mutation to enhance GUI testing coverage", IEEE transaction on Software engineering,vol-1,Issue-2,pp.567-581,Feb 2013.

[10] Xiaoyin Wang, Lu Zhang, Tao Xie, Hong Mei and Jiasu Sun, "Locating need to externalize constant strings for software internalization with generalized string taint analysis", IEEE transactions on software engineering, vol-39, no-4,pp.245-267,april 2013.


 © 2015, IJCSE All Rights Reserved 