2010 11th Int. Conf. Control, Automation, Robotics and Vision
Singapore, 7-10th December 2010

An FSM based GUI Test Automation Model

Yuan Miao
School of Engineering and Science
Victoria University
Melbourne, Australia
yuan.miao@vu.edu.au

Xuebing Yang
School of Engineering and Science
Victoria University
Melbourne, Australia
xuebing.yang@live.vu.edu.au

***Abstract*— Graphical User Interfaces (GUIs) constitute a large proportion of today's software and are becoming more and more complex. Testing the correctness of GUIs and their underlying software is paramount for providing quality software products. Manual testing is extremely slow and unacceptably expensive. We present a new technique which enables the process of generating test cases and testing automation, based on an innovative model. Given a GUI based application, the set of GUI states and their running logic is modeled as a finite state machine (FSM). The efficiency of the model is formally analyzed and compared with event flow graph (EFG) model. The results show that our model is more efficient in storage.**

***Keywords*— GUI Test Automation, Finite State Machine, Event Flow Graph**

I. Introduction

Graphical User Interfaces (GUIs) have become the most popular and accepted way of interacting with today's software. GUI based software includes codes for both implementing GUIs and their underlying functions. GUIs use up to 60% of the total software code [1][2]. Although the GUIs make software easy to use from a user's perspective, they bring a complexity to the software development process, which includes design, coding and testing [3][4]. Testing GUIs is more complex than testing conventional software, as not only does the underlying software have to be tested but also the GUI itself, which is extremely expensive.

GUI test automation is traditionally through Capture and Replay (CR) technique [5][6]. CR tools provide a basic automation solution by recording mouse coordinates and user actions as scripts. A major problem of using mouse coordinates is that the scripts can break with even minor changes to the GUI layout.

To overcome the difficulties associated with recording mouse actions and coordinates, a series of modern CR tools has been developed. Included are the popular Quick Test Professional(QTP), Abbot, Selenium, and Rational Functional Tester(RFT), Win runner, SilkTest, and IBM Rational Robot. These CR and visual test tools capture values of various properties of GUI objects rather than mouse coordinates. The recorded events are connected to GUI objects (widgets such as Textbox, Button, etc.) by using unique names. Unique names can be identified with collections of values of the properties of GUI objects. When interacting with the application, the unique name will be used to obtain the reference to the real object in the GUI and recorded events will be performed on the designated object.

CR tools are very useful but inadequate to perform true automation tasks such as GUI testing. They can only record user actions on the GUIs of the given applications and replay these actions. Test automation needs the knowledge of the logic or workflow of the GUIs. Researchers have developed a series of techniques for GUI test automation ([8]-[12]). However, these proposals are based on manually created models from the application's specifications [13], which involve strenuous labor and intensive resources. This specifications based GUI automation has proven to be impractical and not really feasible [14].

To automate the process of GUI testing, a graph-traversal model, event flow graph (EFG), and its later version, event interaction graph (EIG)[15]-[18] and event sequence graph (ESG) [19][20], have been proposed in recent years to generate sequences of events for creating test cases. The EFG based test automation ([18]) has been claimed as the first practical GUI automated smoke test. This was followed by research on automated black-box GUI testing. In this research, the event flow graph (EFG) was proposed as the core-enabling model. In EFG, each vertex represents an event. All events which can be executed immediately after this event are connected with directed edges from it. A path in EFG is a legal executable sequence which can be seen as a test case. EFGs can be generated automatically using a tool called GUI Ripping ([16]) Traversing the EFG with certain strategy can generate test cases.

EFG was first proposed in [15], the definition of EFG being as follows.

Definition: An event-flow graph for a component C is a quadruple $<V, E, B, I>$ where:

1. V is a set of vertices representing all the events in the component. Each $v \in V$ represents an event in C;

2. $E \subseteq V \times V$ is a set of directed edges between vertices. Event e_i follows e_j iff e_j may be performed immediately after e_i. An edge $(v_x, v_y) \in E$ *iff* the event represented by v_y follows the event represented by v_x ;

3. $B \subseteq V$ is a set of vertices representing those events of C that are available to the user when the component is firstly invoked; and

4. $I \subseteq V$ is the set of restricted-focus events of the component.

In the definition, a GUI component *C* is an ordered pair <*RF, UF*>, where *RF* represents a model window in terms of its events and *UF* is a set whose elements represent modeless windows also in terms of their events. Each element of *UF* is invoked either by an event in *UF* or *RF*.

To generate the test cases automatically, events are classified into 5 groups:

1. *Restricted-focus* events open modal windows;
2. *Unrestricted-focus* events open modeless windows;
3. *Termination events* close modal windows;
4. *Menu-open events* are used to open menus; and
5. *System-interaction* events interact with the underlying software to perform some actions.

To create EFG automatically, finding the follow-up events of each event is critical. It can be done using an algorithm called GetFollows [15].

As the authors claimed, it is vital that an EFG based approach firstly has some practical GUI test automation. For a program like Paint, this model can automatically generate around 8000 test cases and verify them within 10 hours. In this paper, we will further push the frontier of the research by proposing a new model to address a number of limitations of EFG based GUI test automation:

1) The EFG model and its relevant technologies have not been widely accepted by other researchers and software engineers.

2) To execute test cases, EFGs alone are insufficient due to the lack of GUI state information. In other words, a state machine model is still needed to navigate the GUI, retrieve the runtime objects of the GUI and perform the test cases.

3) EFGs are not able to model situations when the underlying code changes the GUI object dynamically, for example:

- Non-fixed events set. GUIs exist in many applications where the visibility of some objects is changed by the underlying code according to another object's state (e.g., the 'Checked' property value of a checkbox). In this case, the event leads to these GUIs being undefined or ill defined in the following event set. They are dependable on the property of the checkbox.
- Expandable panel. Such GUIs also widely exist in many applications such as Microsoft Office 2007, where some panels (or modeless windows) are sometimes visible and sometimes invisible. Toggling the visibility-property value of a panel can cause some events to be exposed or hidden (controls in the newly enabled panel or modeless window). According to the core algorithm, GetFollows, of the EFG model, these events are not able to be modeled, and thus cannot be tested.

Finite State Machine (FSM) is one of the most widely used models in software design and software testing, especially for GUI modeling. This model is embedded in many commercial design and test platforms. It is thus ideal to have an FSM based GUI test automation model. In this paper, we propose an FSM based GUI Test Automation Model (FSM based GuiTam), which can automate all the tests of the EFG based approach. We will prove that the efficiency of the new model, in terms of storage and computational complexity, is at least as good as that of EFG GUI test automation. Furthermore, this model is also able to automate GUI tests in the scenarios of a "non-fixed events set" and "expandable panel", which cannot be addressed by the EFG model.

The rest of the paper is organized as follows: the FSM based GUI Test Model is presented in section II. Section III analyzes the GuiTam in terms of efficiency and applicable cases. Conclusions and discussions are presented in section IV.

II. FSM BASED GUI TEST AUTOMATION MODEL

In this section, we propose an FSM Based GUI Test Automation Model. This model is not only able to achieve the GUI test automation of the EFG model but also able to automate more GUI testing which the EFG cannot do.

A. State of GUIs

There are many ways to define the states of a GUI application. To facilitate the GUI test automation, we shall focus on GUI related state and state transitions. Thus we consider the state of a GUI application as follows: the graphical user interface of a given application is treated as a series of interfaces. Each interface can be regarded as a state. We will use this state to construct a finite state machine for GUI test automation. A GUI's state is modeled as a set of opened windows and the set of objects (label, button, text, etc.) contained in each window. Hence, at a particular time t, the GUI can be represented by its constituent windows

$\boldsymbol{W} = \{w_1, w_2, \ldots, w_n\}$

and their objects

$\boldsymbol{O} = \{O_1, O_2, \ldots, O_n\}$,

where

$O_i = \{o(i,1), o(i,2), \ldots o(i, m_i)\}$, $i=1, 2, \ldots, n$;

each object contains properties

$$\begin{aligned}\boldsymbol{P} = \{\ & P(1,1), P(1,2), \ldots, P(1, m_1), \\ & P(2,1), P(2,2), \ldots, P(2, m_2), \\ & \ldots, \\ & P(n,1), P(n,2), \ldots, P(n, m_n)\},\end{aligned}$$

where

$P(i,j) = \{ p(i,j,1), p(i,j,2), \ldots, p(i,j,k_{ij}) \}$;

$i=1, 2, \ldots, n;\quad j= 1, 2, \ldots, m_i$;

and their corresponding values

$V(i,j) = \{ V(i,j,1), V(i,j,2), \ldots, V(i,j,k_{ij}) \}$;

where

$V(i,j,k) = \{ v(i,j,k,1), v(i,j,k,2), \ldots, v(i,j,k,L_{ijk}), \}$,

$i=1, 2, \ldots, n;$

$j= 1, 2, \ldots, m_i$;

$k= 1, 2, \ldots, k_{ij}$;

At a certain time t, the set of windows and their objects constitutes the state of the GUI. All the objects are organized as a forest. A GUIs state is then modeled as a quadruple (***W,O, P,*** *V*). Events $\{e_1, e_2 \ldots e_q\}$ performed on the GUI may lead to state transitions. The function notation $S_j = e_i(S_i)$ is used to denote that S_j is the state resulting from the execution of event e_i at state S_i . Such a state and transition can be considered as a finite state machine. However, such an FSM would contain too many states and transitions which would make the test automation impractical. For example, the EFG test automation on program Paint has about 8000 test cases, for a maximum of three interactions, and which do not consider all the possible objects, properties and values. Therefore, the cases for full permutation of all the states can easily require over a million years to execute the tests.

B. FSM Based GUI Test Automation Model

An FSM Based GUI Test Automation Model (GuiTam) is a quadruple (Σ, S, s_0, T), where:

- Σ is a finite, non-empty set of all possible input events of the application under test (AUT),
- S is a finite, non-empty set of GUI states of AUT,
- s_0 is an initial state, an element of S, and
- T is the state-transition function set: $T: S \times \Sigma \rightarrow S$.

In this model, all possible events of an AUT constitute the input set Σ, and s_0 is the first state of the AUT when it is invoked. S is composed of all possible states of the AUT. For each $s \in S$ is a tuple $<W,E>$, where W is the object forest and E is the set of possible events in this state, $E \subseteq \Sigma$. For each $t \in T$, $s' = t(s, e)$, where s is the current state, s' is the next state, e is the event in s. We can also simply use $<s,s',e>$ to describe a transition. By performing event e on s, the state will be changed to s'. It is possible for $s' \neq s$, and also for $s' = s$.

Depending on the definition of states, there can be a widely different number of states for the same AUT. For GUI test automation, ignoring all properties will leave many cases where the resulting state of some states is not well defined (not unique). However, if we differentiate all different property values as different states, the number of states is too big to be computationally feasible. In this paper, we use selected objects equivalence classes of the property value set as different states to achieve a good balance. The state selection in the proposed GuiTam is at a similar level to the EFG model, which is also practical in terms of storage and computational complexity.

C. Automatic construction of GuiTam.

Automatically generating a GuiTam requires it to read the widgets (objects) and perform the events on the GUI of given AUTs. We have built a set of fundamental tools that read the widgets (objects), check states and perform events on the GUI. ReadState reads the current state of a given AUT, Existing(*s, S*) tells whether s is contained in S according to certain criteria, GetEquivalentState(*s,S*) returns the state in S which is equivalent to s according to certain criteria, and MoveTerminateEventsToBottom(*E*) moves all the terminate events to the bottom of the events collection, which is used when a new state is found. In each state s, we use a variable '*nextIndex*' to record the next events to be performed. Algorithm 1 can recursively construct the GuiTam for a given AUT.

GuiTam Generation Algorithm

```
1.  AutoGenerateGuiTam (
2.   ps:GUIState,   //the previous state
3.   pe:Event,      //the event performed previously
4.   M:FSM_GUI_TAM
5.  )
6.  {
7.  s=ReadState();//read the current state
8.  if( s=null ) return; //failed, algorithm finishes
9.  if(!Existing(s,S)){
10.    s. nextIndex =0;
11.    MoveTerminateEventsToBottom(s.E);
12.    if(M.S=Ø) M.s_0=s;
13.    M.S=M.S ∪ {s};
14.    M. Σ = M. Σ ∪ s.E;
15. }
16. else{
17.    s=GetEquivalentsState(s,M.S);
18. }
19. if(ps ≠ null && pe ≠ null){
20.    t=<ps,s,pe>;
21. }
22. if(s. nextIndex >s.E.count) return;
23. e=s.E[s. nextIndex];
24. s. nextIndex =s. nextIndex +1;
25. Perform(e);
26. Wait(watitime)
27. AutoGenerateGuiTam (s,e,M)
28. }
```

GuiTam Generation Algorithm is a recursive function. It has 3 parameters: *ps* is the previous state; *pe* is the previously performed events that belong to *ps*. Both *ps* and *pe* are null when the algorithm is firstly invoked. *M* is a GuiTam. *M* is empty when the algorithm is first invoked, and is the final GuiTam when the algorithm finishes.

Line 9 - line 18 check whether the current state is a new state or not. If it is a new state, the state is initialized by setting the next performing event index to 0, moving the terminate events to the bottom of the available events collection and adding the state to the collection of *M*. If it is not a new state, the equivalent state is found, which was previously stored in the *M*.

Line 19 - line 22 add a transition to *M*

Line 23 – line 25 prepare the next event in current state to perform. If no more events to perform, finish the function.

Line 28 recursively invokes the algorithm itself to check the next state.

The output of this algorithm is an FSM based GuiTam for the given AUT.

An example GUI flow part of a clinical software, its object forest (captured from the system) and the state transition model can be found at:

http://miaoy.awardspace.biz/research/GuiTam.htm.

D. GUI Test Automation :Test cases

With the FSM based GuiTam model, e.g. *M*, test cases can be automatically generated by traversing the states in *M*. The transitions in *M* can be seen as directed arcs. By using directed graph depth-first traverse algorithms, combined with certain criteria, test cases can be generated automatically. Note that, unlike a traditional directed graph, there may exist more than one transition from one state to another (even to itself), which means more than one arcs from one state to another.

E. GUI Test Automation: Test oracles

A test oracle for a test case, $e_1, e_2, \ldots, e_n$, where each e_i is an event, is represented as a sequence of states $s_1, s_2, \ldots, s_n$. Depending on the criteria of the given AUT, each state contains the complete or partial state of the real GUI after an event is performed. The oracle information will be captured from the original version (before modification) by performing the same test case on the original version. After the test oracles are generated, the test cases can be executed on the modified version of the AUT and can compare the state information of the modified version with the corresponding oracle information. If there are differences, possible defects are discovered.

III. Analysis of FSM based GuiTam

In this section, we will analyze the FSM based GuiTam as compared to the EFG test automation. We will prove that for each EFG based test, there exists a GuiTam that can automate the test, with no more requirements on the storage and computational power. We will also illustrate that for the two scenarios of "non-fixed events set" and "expandable panel" presented in section I, GuiTam is able to automate the test while EFG cannot.

For the convenience of comparison, the core algorithm GetFollows ([14][15]) of the EFG based GUI test automation is shown as follows.

Algorithm GetFollows

```
1.  GetFollows( v: Vertex or Event)
2.  {
3.     if(EventType(v)=menu-open ){
4.        if v ∈ B of the component that contains v
5.           return ( MenuChoices(v) ∪ {v} ∪ B);
6.        else
7.           Return
8.              (MenuChoices(v) ∪ {v} ∪ B ∪ (Parent(v)));
9.  }
10. if(EventType(v)=system-interaction)  return (B);
11. if( EventType(v)=termination)
12.       return (B of Invoked component);
13. if (EventType(v)=unrestricted-focus)
14.       return   (B ∪ B of Invoked Model Dialogue)
15. if (EventType(v)=restricted-focus)
16.       return (B of Invoked component);
17. }
```

The details of the algorithm can be found in [28][29].

A. Inclusive Mapping Between EFG and GuiTam

In this section, we will prove that there exists an inclusive mapping between EFG and GuiTam, that is, for each EFG, there exists at least one GuiTam that is able to automate all the EFG automated tests.

Theorem 1: For each EFG, there exists at least one GuiTam, which can automate the tests of the EFG.

Proof: Let $C = <V, E, B, I>$ be an EFG. Let $\Sigma=V$ be the input domain which contains all the possible events. Because an event is always related to a GUI object, we can generate a state from any set of events. Suppose there is a function *GenState(X)* which can generate a state from a set of events *X*. S_0*=GenState(B)*. $s_0.E=B$ is the initial state which contains the set of events of *C* that are available to the user when the component is first invoked; $T = S \times \Sigma \rightarrow S$ is a set of transitions $t=<s,s',v>$, where $s,s' \in S$, $v \in s.E$. s' is the state when $s'.E=\{v' \mid <v,v'> \in E\}$. Starting from s_0, all the transitions and their next states can be generated recursively. Thus, GuiTam $M=< \Sigma,S,s_0,T >$ can automate all tests of the EFG.

The proof of Theorem 1 also provides an approach to convert an EFG into an FSM. A drawback of this method is that there are often a number of unnecessary states in *S*, i.e., many states in *S* contain the same set of events. The number of states equals the number of events in Σ. The number of transitions in *T* equals the number of events in Σ as well. If we unite all the states, i.e., those states with the same set of events will be considered as one state, the number of states will be greatly reduced. EFG's Inclusive GuiTam Algorithm provides an algorithm to construct a more effective GuiTam of a given EFG.

EFG's Inclusive GuiTam Algorithm

```
1.  Transform(efg: EFG,M: FSM_GUI_TAM)
2.  {
3.     M.Init();
4.     M. Σ=efg.V;
5.     M. T = Ø;
6.     M.s0=GenState(B);
7.     M.S = M.S ∪ s0;
8.     Convert(efg, M, s0);
9.  }
10. Convert(efg: EFG, M : FSM_GUI_TAM,s:State)
11. {
12.   For each v in s{
13.      X={v' | <v,v'> ∈ efg.E} ;
14.      s'=GenState(X);
15.      s'.E=X;
16.      t=(s,s',v)
17.      M.S= M.S ∪ {s'};
18.      M. T = M. T ∪ {t};
19.      M. ∑= M. ∑ ∪ X;
20.      Convert(efg, M, s');
21.   }
22. }
```

B. Storage Analysis

To automate the GUI test that an EFG is able to perform, the GuiTam needs less storage than that of EFG. The storage analysis is provided in Theorem 2.

Theorem 2: the space requirement of *M* created by algorithm 3 is not greater than the given EFG.

Proof:

In the given EFG, vertices: = number of events. The order of storage requirement is *O(n)*.

Different types of events have a different number of edges (*B* is defined in EFG):

- Menu-open: let *l*= number of events in *B* in EFG, m=number of menu-choices, the number of edges for these events is $C^2_l + m = O(l^2)$.
- System-interaction: let l=number of events in B, then the number of edges for these events is $C^2_l = O(l^2)$.
- Termination: let k=number of events of B of invoked components, then the number of edges for these events is k. The order is *O(k).*
- Unrestricted-focus: let l = number of events in B, j = number of events in invoked modalless window, then the number of edges of the events is $C^2_{l+j} = O((l+j)^2)$.
- Restricted-focus: let q=number of events in invoked modalless dialog, and then the edges of these events are q. The order is *O(q).*

Because the number of events in B is much larger than the number of events of termination and invoking modal dialog, so in each component, the number of events in B is very close to the total number n. On average, the number of edges in EFG is $O(n^2)$.

In the constructed M, according to the definition of EFG, every event has a fixed set of follow-up events, which means the values of properties are ignored. The number of states in M is approximately equal to the number of windows and possible popup menus, which is much less than the total number of events. In one component C, the state is about the number of top level menu items plus one.

Number of inputs in Σ equals the number of events, which is of *O(n).*

Number of transitions T equals the number of events, which is of *O(n).*

On average, the space complexity of M is of *O(n),* which is one order less than that of the given EFG.

C. Computational Complexity Analysis

Both EFG and GuiTam can be presented as directed graphs. Once the test cases are generated, the execution of the tests is same with both models. So we just need to analyze the computational complexity for generating test cases from the models. The computational complexity of generating test cases depends on the requirement coverage of the test cases, the length of each test case and the number of the test cases to be generated. A test case is a events sequence $(e_1, e_2, ..., e_k)$ where k is the length of the test case.

Theorem 3: Given the same length of each test, the computational complexity of generating test cases from the two models are of the same order.

Proof: Let n be the number of all events, and k be the test case length.

In the given EFG, except for a small number of events, most of the events have edges between each other, so the directed graph can be seen as a complete connected graph. The edge number of the graph $q = C^2_n$. To generate the test cases, traverse the graph and collect all the cases with length k, then the computational complexity is C^k_n.

In GuiTam, each edge is related to an event, so the number of edges is the n. To create a test case with length k, k edges need to be selected from the edge collection. The total number of possible combinations is C^k_n.

Thus, the computational complexity of generating test cases for both of the two models is of the same order.

D. Dynamic GUI Interactions

According to the definition of EFG and the algorithm GetFollows(Algorithm 2), we can see that EFG cannot model dynamic GUI interactions. There are two major GUI interactions which EFG fails to model but which GuiTam can. We use two example scenarios to illustrate this.

- *Scenario1: Non fixed events set.*

We show an example code first (suppose B1Click is in Form *f1*):

```
1. B1Click()
2. {
3.    Form2 f2=new Form2();
4.    if(checkBox1.Checked)  f.BAccept.Visible=true;
5.    else f.BAccept.Visible=false;
6.    f.ShowDialot();
7. }
```

From this piece of code, according to the definition of EFG, and the algorithm GetFollows, will there be an edge from event B1Click to BAccept in Form *f*? If there is an edge, then when 'Checked' is false, executing the edge leads to failure. If there is not an edge, when 'Checked' is true, this case will never be executed. EFG cannot model this scenario.

With the GuiTam model, we take the value of the 'Checked' property of checkBox1 to differentiate states, so the form *f1* will have states which are called stateChecked and stateUnChecked respectively. *f2* has two states as well, named as stateWithAccept and StateNoAccept respectively. Then we can give the states graph as figure 1.

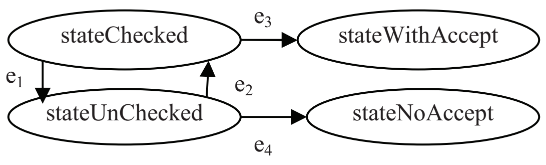

Figure 1. Non fixed events set in FSM based GuiTam

In Figure 1, two transitions, e_1 and e_2 are related to the checkBox1Changed event.

e_1:<stateChecked, stateUnChecked, checkBox1Changed>

e_2:<stateUnChecked, stateChecked,checkBox1Changed>

e_3:<stateChecked, stateWithAccept,B1Click>

e_4:<stateUnChecked, stateNoAccept,B1Click>

It can be checked that all the execution flows of the application are modeled and this can be automated by the GuiTam.

- *Scenario 2: Expandable panel*

We show a piece of code first (suppose there are two panels, panel1 is always visible, panel1 has two buttons: B1 and B2. Panel2 can be either visible or not, affected by the event of B1Click. Penel2 has two buttons B3 and B4. The two panels are in one form.).

```
1.  B1Click()
2.  {
3.      panel2.visible=!panel2.visible;
4.  }
```

In EFG, event B1Click has uncertain follow-ups. According to the algorithm, GetFollows can have different results and thus is undefined after B1Click, if panel2 is visible, only B3Click and B4Click are connected to B1Click. However, a lot more needs to be done:

- B3Click, B4Click should be connected to B2Click
- B1Click, B2Click should be connected to B3Click
- B1Click, B2Click should be connected to B4Click

EFG are not well defined in this situation.

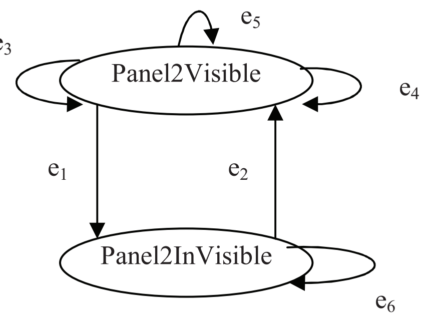

Figure 2. Panel visible changes state in FSM GUI TAM

In GuiTam, the two possible values of panel2's property 'Visible' are seen as two different states. Figure 2 illustrates the states and the transitions. The transitions are:

- e_1 <Panel2Visible, panel2Invisible,B1Click>
- e_2 <Panel2InVisible, panel2Visible,B1Click>
- e_3 <Panel2Visible, panel2Visible, B2Click>
- e_4 <Panel2Visible, panel2Visible, B3Click>
- e_5 <Panel2Visible, panel2Visible, B4Click>
- e_6<Panel2InVisible, panel2InVisible, B2Click>

IV. Conclusions and Discussions

In this paper, we have presented an innovative model based on finite state machine to automate the process of GUI testing. The model is able to automate all the GUI tests that can be automated by the EFG model. We have proved that the storage requirement of the GuiTam is one order less than that of the EFG model, and the computational complexity is at a similar level. For the two commonly used GUI components of non-fixed event sets and configurable panels, GuiTam is able to automate the test while EFG cannot model such GUI applications. This paper has also presented algorithms to automatically construct GuiTam for any given EFG to GUI test automation.

References

[1] Brooks, P., Robinson, B., and Memon, A. M., "An initial characterization of industrial graphical user interface systems," in ICST 2009: Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation. Washington, DC, USA: IEEE Computer Society, 2009.

[2] Myers B. A. User interface software tools. ACM Transactions on Computer–Human Interaction 1995; 2(1):64–103.

[3] Myers B. A., "Why are Human-Computer Interfaces Difficult to Design and Implement?", Technical Report CS-93-183, School of Computer Science, Carnegie Mellon Univ., July 1993.

[4] Wittel, W.I. and Lewis, T.G. "Integrating the MVC Paradigm into an Object-Oriented Framework to Accelerate GUI Application Development," Technical Report 91-60-06, Dept. of Computer Science, Oregon State Univ., Dec. 1991.

[5] Anderson, J. and Bache, G., "The video store revisited yet again: adventures in GUI acceptance testing," in Proceedings of the 5th International Conference on eXtreme Programming and Agile Processes in Software Engineering, LNCS 3092, 2004, pp. 1-10.

[6] Dutta, S., "Abbot – A friendly JUnit extension for GUI testing," Java Developer Journal, Vol. 8, 2003, pp. 8-12.

[7] McMaster, S., Memon, A. M., "An Extensible Heuristic-Based Framework for GUI Test Case Maintenance," icstw, pp.251-254, IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2009

[8] Shehady, R. K. and Siewiorek, D. P. 1997. 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). IEEE Press, Washington - Brussels - Tokyo, 80–88.

[9] White, L. and Almezen, H. 2000. Generating test cases for GUI responsibilities using complete interaction sequences. In ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00). IEEE Computer Society, Washington, DC, USA, 110.

[10] Offutt, A. J. and Hayes, J. H. 1996. A semantic model of program faults. In ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis. ACM Press, New York, NY, USA, 195–200.

[11] Ana C. R. Paiva, Nikolai Tillmann, João C. P. Faria, Raul F. A. M. Vidal. Modeling and Testing Hierarchical GUIs in Proceedings of the 12th International Workshop on Abstract State Machines, pp.-, 2005

[12] Ana C. R. Paiva Automated GUI Testing in Informática, XIII Convención Y Feria Internacional, pp.-, 2009

[13] Yuan, X., Memon, A. M.., 2010, Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback, IEEE Transactions on Software Enginerring. Vol. 36, no. 1, January/February 2010.

[14] Memon, A. M.. "An Event-Flow Model to Test EDS". In Software Engineering and Development, (Enrique A. Belini, ed.), 2009.

[15] Memon, A. M., Soffa, M. L., Pollack, M. E., 2001, Coverage Criteria for GUI Testing. Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, Vienna, Austria. Pages: 256 – 267. 2001.

[16] Memon, A. M., Banerjee, I. and Nagarajan, A. " GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," Proc. 10th Working Conf. Reverse Eng., pp. 260-269, Nov. 2003

[17] Xie, Q. and Memon, A. M. 2006. Automated model-based testing of community-driven open source GUI applications. In ICSM '06:

Proceedings of the 22nd IEEE International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, 145–154.

[18] Memon, A. M. and Xie, Q. 2005. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. IEEE Transactions on Software Engineering 31, 10 (Oct.), 884–896.

[19] Zhu, H., Wong, W. E., Belli, F. Advancing test automation technology to meet the challenges of model-driven software development: report on the 3rd workshop on automation of software test, ICSE, 2008

[20] Belli, F. Finite-State Testing and Analysis of Graphical User Interfaces, ISSRE, 2001.