USING XML FOR USER INTERFACE DOCUMENTATION AND DIFFERENTIAL EVALUATION

[1]MOHAMMAD TUBISHAT, [2]IZZAT ALSMADI, AND [3]MOHAMMED AL-KABI

[1]Lecturer, Computer Science Department, Yarmouk University, Irbid, Jordan

[2]Asstt. Prof., Computer Information Systems Department, Yarmouk University, Irbid, Jordan

[3]Asstt. Prof., Computer Information Systems Department, Yarmouk University, Irbid, Jordan

E-mail: mtubishat@yu.edu.jo , ialsmadi@yu.edu.jo , mohammedk@yu.edu.jo

ABSTRACT

In any software product, the user interface is one of the most important parts that defines the communication between the user and the system. One of the challenges in user interfaces is in the ability to save its state at any time as in many cases, especially in problems such as power interruptions; there is a need to save the last “good” state. In some other cases, there is a need to see if the user interface state is changed or to compare the current state with a previous one. This is required for several possible cases such as: evaluation, and changes in requirements, design and implementation. The format used for storing the state of the Graphical User Interface (GUI) should be global and can be easily used by different types of applications (such as the XML format). A GUI state is usually defined as the overall combination of properties of all the components or widgets of the GUI. In this paper an alternative back end representation is proposed for user interfaces and their state from their original format within the applications. In this approach the user interface is converted to an XML file. This XML file represents the current state of the user interface. To avoid state explosion, in this representation the contents of the state considers only the structure of the user interface and ignores controls’ properties that are state irrelevant. A control property is considered as state relevant if it may cause a state change if its property is changed. For example, in a GUI control, the control name, ID, tag, etc are usually irrelevant to the GUI state change compared to the control location, color, or the “visibility” and “enable” properties which are state relevant. The XML file format is largely used and accepted by many software applications. It is the infrastructure language for web pages and database management systems. User interface documentation is useful for future evaluation and comparison and useful for stakeholders’ communication. In this research, it is also proposed for GUI testing activities such as regression testing where testing is triggered if there is a change in the state of the user interface. A tool is developed to automatically evaluate the possible changes of the user interface d. Several open source projects are used for testing and evaluation. In each case, different types of GUI state changes are designed, tested and evaluated.

Keywords: *GUI States, User Interface Testing, Modeling, XML, And State Charts.*

1. INTRODUCTION

In modern software applications and every new software products, one of the most important parts of the software is the user interface. In many cases we are required to change the user interfaces as requirements changed. There are many ways that you can use to document the user interfaces such as prototypes, screenshots, descriptions, etc. In many applications we use the user interface saved state for many operations in our software like restore, redo, and undo. State charts are used to generate tests in model-based architectures. They describe state transitions of objects with states. Model checking is a technique used for verifying a system composed of concurrent finite-state machines. State machines should be finite as model checkers need to exhaustively explore the entire state space of the state machine.

In user interface testing, the general accepted approach defines a GUI state through all GUI components and their properties. This means that a change in a single property of a single GUI control or widget causes the GUI state to be changed [1, 2, 3, 4, 5, 6, and 7]. Defining the GUI states by the combination of all its widgets and properties produces a very large number of possible GUI states. Changing any single property in any widget causes the whole GUI state to be different. A widget is a graphical user interface element responsible for interacting with the user. There are some variations in the literature between the meaning of widget and control. Here we will assume that they have the same meaning.

In modern applications, the ability to save and control the GUI state is very useful in several features such as: Undo, redo, animation, hide, show, enable, disable, etc. For example, a user wants to be able to undo an action or actions and remove their effect all over the application. If a user is using an application and power failure is suddenly occurred, the application should have the ability to save the last GUI state to allow it to be recovered. The knowledge of GUI state changes can be used for several activities such as: restore, backup processes and regression testing.

In this project, we propose a different way to define the GUI state. We suggest defining the GUI state through the XML file that represents the GUI. The reason for doing this is to automate the way we compare two different GUI states (through XML files' comparison) and to save the GUI state in a universal format that can be stored for later stages in the projects or transferred to another application.

When using XML to store the GUI tree, a new definition is introduced for a GUI state. Rather than assuming that the GUI state depends on each property for each control of the whole application, we define the GUI state as the hierarchy that is embedded in the XML tree. A GUI state change here means a change in any parent-child relation, or any change in the specific properties that are parsed. This new definition produces an effective reduction in GUI states. For example a small application such as Notepad, can have more than 200 controls, each of which may have more than 40 properties, this produces 200 X 40 = 8000 states. Any property or control from the 8000 controls and properties triggers the state change. In our assumption, 200 controls will have 200 parent-child relations (or less). Each control has less than 10 relevant parsed properties. The total possible GUI states are reduced to 2000 or a 75% total reduction. For example, if we take a class in a university system as an example and consider the states as: new, occupied and full class, then the only property that will cause a state change in the class table is the number of registered students. The name of the instructor, the class location, etc will not be relevant to this state as they will not cause a state change once their value is changed.

In testing, we usually use the evolution or modification of the user interface state as a reason to trigger or reapply several testing processes. For example, In GUI test automation, GUI scripts should be regenerated or revisited upon GUI state changes (regression testing). As such activities can be expensive; we need to trigger them only when there is a good reason to do that.

Each GUI has many forms, each form has many controls, and each control has many properties. The general definition used in many literatures [1, 2, 3, 4, 5, 6, and 7] that a change in any control property causes a GUI state change means that the number of states in any GUI, is very large even for a small application.

The GUI state change is used in testing to trigger some testing activities such as integration or regression testing. The number of possible GUI states affects the space of creating test cases which means that the more possible states we have, the more test cases we need to generate for test adequacy.

2. RELATED WORK

An object state is the condition(s) of that object in a given time. A given state for an object defines the events that may affect it at this time, and the next possible states. A GUI state is described in terms of the specific objects or controls it currently contains, and the values of their properties [8, and 9]. The information of a GUI state at any particular time is important for testing.

Saving the GUI to a GUI state file is investigated by several papers [10, and 11]. The GUI state can be saved and retrieved from such files. This facilitates the usage of some services such as: undo, simulation, and testing, and the storage of the current state.

In literature, usually, there is an ambiguity between the application and the user interface states. Despite the fact that those two states are related, yet they are not identical. The application state is the state of all the application resources, including the user interface, at any given time.

Currently, there are many user interface description languages used to facilitate communicating with the interface, and the code that sets behind it. This is occurred in different software engineering tasks such as: requirement, design and testing. We will use XAML as an example. XAML (eXtensible Application Markup Language; pronounced "Zammel") is a new Microsoft Longhorn declarative markup programming language for building applications' user interfaces. Elements in the XAML file are correlated to the GUI objects at run time. XAML utilizes XML hierarchical logic to present the hierarchy of GUI objects. This makes our testing framework matches in principles the approach XAML is taking. It is possible to develop a solution with XAML without developing any code or develop partial XAML/code applications. XAML technology is presented here as related work as it is an example of a user interface description language or a tool to document the user interface description.

The ultimate goal of XAML is to have a standard syntax for describing user interface controls and eventually serializing all GUI components in XAML files. This will be very useful in several ways. In one particular advantage, the UI implementation will be documented in a way that can be easily accessed, edited, and reconstructed. Some other advantages expected from having a standard syntax for UI design and implementation is; the ease of modifying the UI even at run time, the ability to separate UI components from the other layers of the application or separate the UI model from its view, and the ability to reuse the UI or some of its components.

XAML can be used as a UI modeling language to create UI elements that can be implemented in any platform and with any programming language (theoretically). XAML simplified control properties into its type and text only. The created button has default visual presentation through theme styles, and default behaviors through its class design. For testing, this reduces the large amount of possible UI states relative to alternatives.

A control can be presented in XAML using the line;
<Panel1> <Button Content="OK"/> </Panel1>, where the panel, panel1 and the button OK are two XAML elements.

XAML elements (i.e. UI controls) are mapped to .NET types that can be extracted from their assembly. Abstract classes are not mapped to XAML tags.

The second related subject we will discuss is Avalon and Windows Presentation Foundation layer (WPF). WPF is WinFX (i.e. .NET framework 3.0) user interface framework or graphics subsystem platform for Windows client's applications. It is preinstalled in MS Windows Vista operating system. In WPF, control's logic is separated from its appearance that adds flexibility to the way controls can be displayed. WPF content can be hosted in a Win32 window and visa-versa.

There are some commercial and open source GUI builder tools used specially for Rapid Application Development (RAD) in which those tools can generate and design user interfaces and save them in XML files to be accessed and utilized by an application that can read this universal storage format. Examples of such tools are: Design In Real Time (Dirt) [14] and Glade [15]. For example, in Glade, similar to the tool we developed in this project, all user interface widgets structures, the properties of each widget and any sign handlers associated with them are parsed to XML files [16]. XML files can be then loaded into the application with the help of another tool called "Libglade". This makes your user interface independent of a programming language. Moreover, often it may allow you to modify your user interface without the need to modify your program and for those that use compiled languages without the need to recompile your application.

In more recent papers Qian et al proposed an event interaction framework to improve GUI testing coverage [17]. The framework or structure is similar in principle to any Finite State Machine (FSM) system that describes GUI possible events and transitions.

Yokoyama et al used model checking to verify executable Java programs through extending Java PathFinder [18]. They proposed three GUI functions that focus on GUI possible states and transitions. Ganov et al proposed using symbolic execution in GUI testing with the goal of optimizing the selection of test cases to generate and improve coverage [19].

Yuan et al utilized run time information to improve GUI testing coverage [20]. The technique alternative the selection of the next batch of test cases based on the knowledge gained from the most recent generated ones. The test cases are generated based on the GUI event graph that is generated from the actual GUI using reverse engineering methods.

User Interface Builders

The earlier goal of the developed tool was to save the GUI of an existed application into an XML file. The second goal was to integrate using the same tool as GUI builder in which the tool will be used to create the XML file from the GUI prototype. This task usually occurs in early stages of development in which user interface designers are willing to store their user interface design in more than simple screen shots. There are many tools that can be used for UI graphic builder. Most widely used IDEs such as .NET, Eclipse, NetBeans can and is capable to rapidly design and develop a user interface. However, the goal here is to store this design in a universal format where this design can be easily transferred from one platform or environment to another. XML files are a suitable for this purpose as they are widely acknowledged and used documentation format over the web and the different environments.

Similar to the steps used in the first stage of the tool, we utilized the fact that .NET managed code store all user interface component information in its assembly. As such, this approach is developed using .NET IDE where the application is compiled and build after developing the user interface in order to store all user interface component information in the assembly. Reflection is then used by the tool to serialize all GUI components, with their properties and relations into an XML file. Such feature can be easily integrated with the IDE to offer a user interface storage in case users wants to permanently store it. Developers of IDE may not need to use reflection to regenerate the components from the assembly as this information is available before that.

3. GOALS AND APPROACHES

For software testing, the reason or goal for studying the state of an object at any given time is to know the “scope” of that object, i.e., to know the current actions that may affect the object and the results of those actions. This is important in particular, for transactional processing applications. For example, if a car is in an initial complete “off” state, some actions such as “switch on” are available, while others, such as “accelerate” are not.

Should changing the color of one control in the whole GUI causes a change to the whole state of the GUI? In other words, will such action disable some events and enable some others? Maybe it is inaccurate to say that in all cases, such change will not have any impact on the overall GUI state, but for the most cases, it will not. To deal with the problem of having large number of GUI states, we have to consider the major ones only. This customized definition of GUI states is suggested in a GUI test automation framework [12, 13]. The application displays the GUI hierarchy, its controls, and properties.

The tool checks for GUI state changes through comparing the current XML tree file that represents the GUI of the application with the previous one (or any other selected one). The tool can display the GUI hierarchy, its controls, and properties. Testers can then specify the properties that they want them to trigger a GUI state change as this can be different with the different scenarios.

The GUI states comparison is done automatically through the tool. This comparison and checking of the overall GUI state (i.e. GUI structural state) is not intended to be used for test case generation. It can be used to trigger the execution of regression testing (In the same way a sensor triggers switching on, or off, an air condition system once it reaches a cut off ,high or low, degrees).

Implementing some actions such as undo, redo, restore, etc using XML files comparison can be straightforward. This means that we can utilize several other advantages of using XML files to represent the GUI structure. Besides using it for undo, redo, or restore actions, this GUI documentation can be used in regression testing or testing in general. It can be also used in future projects earlier stages of requirements and design. One of the challenges in software design and coding is that, especially later in development, developers and other project team members need to have a common ground or form of the application user interface to use for discussion and feedback.

The application user interface can be saved in those files for re-usage and testing. GUI re-use is usually out of context for software developers. However, XML files abstract the GUI structure and save it in a format that can be reused.

Comparing the GUI design and implementation can be achieved automatically if we have the user interface represented through some XML files. Those are some of the advantages sought in using GUI description languages such as XAML and XUL. XUL is an XML based user interface description language used to describe the windows layout. The Internet browser Firefox 3.0 is built using XUL and provides XUL runtime environment.

Implementation and Experimental Work

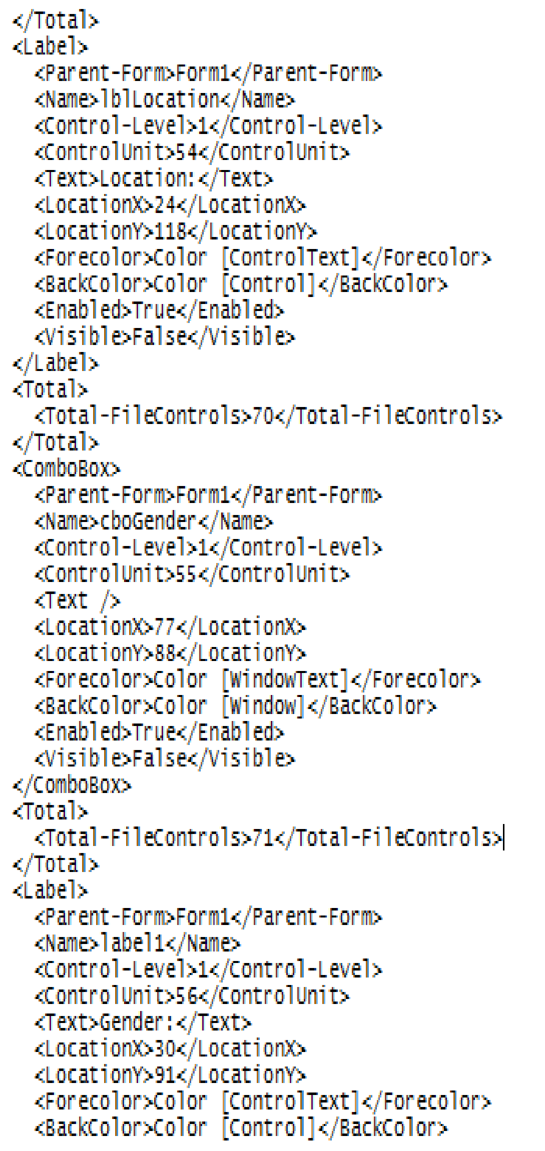
An application GUI can be formally defined by:

$$G = (C,A,V,E,N, X)$$

Where C represents all GUI control components (whether they are containers such as forms, panels, frames, groupboxes, or individual components such as: buttons, textboxes, labels, etc.). E represent the Edges between components where there are certain – definite – number of edges. Each edge connects two consecutive components. “A” represents controls’ attributes. “V” represents values of those attribute. Each control can be distinguished by the attributes and their values. N represents the GUI entry points. In most cases, it maybe denoted by “n” to indicate that there is only one entry point. X represents the exit points. There are some controls that are “leaf” controls. Those controls are not parents for any other control which make them candidate exits. Experimental tests will be developed to change any representative from those six parameters and evaluate the GUI states’ verification algorithms to detect those changes.

In order to implement the proposed approach, a modification is implemented in a previously developed test automation tool [13]. The modification was to serialize all user interface components and properties into an XML file. This method is somewhat similar to the two technologies presented in the literature (XAML and XUL) where the application user interface is described using an XML file. Figure 1 shows a small sample of an XML file generated by the developed tool for a tested application. In the generated file, the following information can be found:

- All user interface controls or widgets with their hierarchical structure preserved according to their location in the user interface. For example, a high level or entry control will exist in the level nodes in the XML file. All controls that are accessed through this component will be followed in the next level and so on. This was one of the reasons for selecting the XML to be the source of storage for the user interface components as XML files by default are of hierarchical nature.
- In each control or widget, all widget attributes along with their values are serialized and preserved.



```
</Total>
<Label>
  <Parent-Form>Form1</Parent-Form>
  <Name>lblLocation</Name>
  <Control-Level>1</Control-Level>
  <ControlUnit>54</ControlUnit>
  <Text>Location:</Text>
  <LocationX>24</LocationX>
  <LocationY>118</LocationY>
  <Forecolor>Color [ControlText]</Forecolor>
  <BackColor>Color [Control]</BackColor>
  <Enabled>True</Enabled>
  <Visible>False</Visible>
</Label>
<Total>
  <Total-FileControls>70</Total-FileControls>
</Total>
<ComboBox>
  <Parent-Form>Form1</Parent-Form>
  <Name>cboGender</Name>
  <Control-Level>1</Control-Level>
  <ControlUnit>55</ControlUnit>
  <Text />
  <LocationX>77</LocationX>
  <LocationY>88</LocationY>
  <Forecolor>Color [WindowText]</Forecolor>
  <BackColor>Color [Window]</BackColor>
  <Enabled>True</Enabled>
  <Visible>False</Visible>
</ComboBox>
<Total>
  <Total-FileControls>71</Total-FileControls>
</Total>
<Label>
  <Parent-Form>Form1</Parent-Form>
  <Name>label1</Name>
  <Control-Level>1</Control-Level>
  <ControlUnit>56</ControlUnit>
  <Text>Gender:</Text>
  <LocationX>30</LocationX>
  <LocationY>91</LocationY>
  <Forecolor>Color [ControlText]</Forecolor>
  <BackColor>Color [Control]</BackColor>
```

Figure 1. A sample output from an XML file generated to serialize user interface components and their attributes.

Here are explanations of the possible changes that may occur in a GUI component that may cause a GUI state change. Those types of modifications in the user interface can be detected by the developed tool through the XML file:

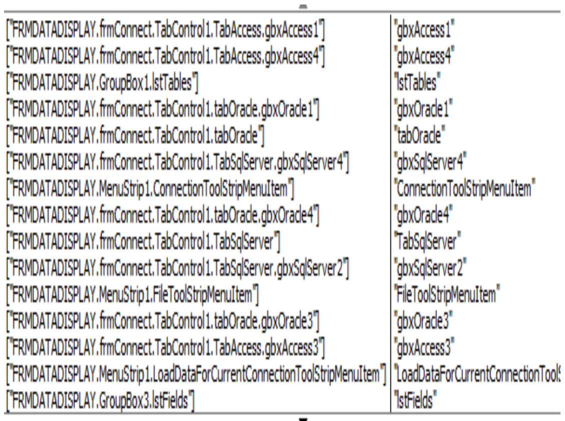
1. Controls [C]. The comparison between two XML files (e.g. original file saved to preserve the user interface state, and a new XML file just serialized dynamically from the user interface) should be able to detect if one of the controls is missing, added or if its location is changed relative to the original file. We developed three algorithms for every one of these three types of modifications (i.e. removed, added or updated control). In all

those cases, user interface state will be changed if one of its components is removed. It will be also changed if a new component is added. Finally, it will also be changed if one of the components changes its location. For example, in MS Word, if the command “Zoom” is moved from the View menu to the File menu, this should cause all MS Word GUI state to be modified. Adding, removing, or updating a GUI component usually occur at design time and rarely occur dynamically or at run time. We noticed that through the very small percentage of this type of change once a large number of real time application and tested and evaluated for types of GUI state changes.

2. Controls’ Attributes [A]. The application user interface will be also changed if a component attribute is added, removed, or modified. This may also rarely happened at run time. The main goal of developing our XML user interface comparison is to enable users to dynamically evaluate if a user interface is changed or not without the need to do this manually which will be a very cumbersome task.

3. Attributes’ values [V]. The third type of user interface state change occurred when at least one attribute value of at least one control is changed. This means that the focus here is in the values of the attributes. The majority of dynamic state changes occurred as a result of such types of actions. However, many may argue that a value change of an attribute should not cause a state change. For example, if the *location* of one component in one form of the user interface is changed vertically or horizontally, should this be considered as a state change? For many reasons, we want to consider this as a state change, specially where a test automation tool is used to test such user interface. In such scenario, the test automation tool needs to know that the location of this control is modified and that it needs to accommodate for this change and expect it in the new location. However, some other control attributes’ modifications such as the modification of the text of a textbox control is less important for the test automation tool to know and accommodate for. To simplify the process, we considered any value modification a trigger to assume a GUI state change. In many cases, a better algorithm should be developed to reduce GUI states’ explosion in which minor state changes such as the one mentioned earlier can be ignored.

Edges [E]. As explained earlier, An edge is an event connection between controls that shows reach-ability between them. A GUI path can be defined as several edges that starts from an entry point and ends in an exit or leaf point. Figure 2 shows a sample output (generated by the developed tool) from GUI paths along with their leaf control names. Each two consecutive controls in the path are connected to represent an edge. This method of defining each control by its unique path is used to check for GUI state change.



```
["FRMDATADISPLAY.frmConnect.TabControl1.TabAccess.gbxAccess1"]             "gbxAccess1"
["FRMDATADISPLAY.frmConnect.TabControl1.TabAccess.gbxAccess4"]             "gbxAccess4"
["FRMDATADISPLAY.GroupBox1.lstTables"]                                     "lstTables"
["FRMDATADISPLAY.frmConnect.TabControl1.tabOracle.gbxOracle1"]             "gbxOracle1"
["FRMDATADISPLAY.frmConnect.TabControl1.tabOracle"]                        "tabOracle"
["FRMDATADISPLAY.frmConnect.TabControl1.TabSqlServer.gbxSqlServer4"]       "gbxSqlServer4"
["FRMDATADISPLAY.MenuStrip1.ConnectionToolStripMenuItem"]                  "ConnectionToolStripMenuItem"
["FRMDATADISPLAY.frmConnect.TabControl1.tabOracle.gbxOracle4"]             "gbxOracle4"
["FRMDATADISPLAY.frmConnect.TabControl1.TabSqlServer"]                     "TabSqlServer"
["FRMDATADISPLAY.frmConnect.TabControl1.TabSqlServer.gbxSqlServer2"]       "gbxSqlServer2"
["FRMDATADISPLAY.MenuStrip1.FileToolStripMenuItem"]                        "FileToolStripMenuItem"
["FRMDATADISPLAY.frmConnect.TabControl1.tabOracle.gbxOracle3"]             "gbxOracle3"
["FRMDATADISPLAY.frmConnect.TabControl1.TabAccess.gbxAccess3"]             "gbxAccess3"
["FRMDATADISPLAY.MenuStrip1.LoadDataForCurrentConnectionToolStripMenuItem"] "LoadDataForCurrentConnectionTool
["FRMDATADISPLAY.GroupBox3.lstFields"]                                     "lstFields"
```

Figure 2. GUI paths sample for an AUT.

4. Entry points [N]. In many cases, there is only one entry point to a desktop or web application. For desktop applications, this is usually the startup form that is called by the Main method. For web applications, this is the homepage for the web site or application. The importance of knowing the entry point is that it is the entry to access all controls, all edges and is considered as the parent of all parents in the application. This is why all GUI paths in Figure 2 starts with “FrmDataDisplay” which is the entry form.

5. Exit points [X]. Unlike entries, exist are usually many. Figure 2 shows many leaf controls that can be considered as exit points for this application. The algorithm that is developed to locate all leafs searches for all controls that are not parent of any other control.

The different types of GUI state events checking (i.e. adding, updating, and removing a GUI state) are developed and applied on several open source projects. First, the original GUI is added and saved for comparison with GUI state changes. The “Add” method is responsible for adding the GUI reference state file. A software development team who is working on continuously and iteratively developing an application in general and its user interface in particular should keep an agreed upon fixed state of the user interface that will be referred to whenever a process needs to know whether a state change occurs or not. For example, regression testing is triggered every time a state changed is occurred. The regression testing database will be executed to make sure that such state change did not cause any problems.

The process of saving a GUI state is implemented through using reflection to serialize or parse all GUI components and their relevant properties and attributes. The hierarchy of the GUI is preserved through studying the application through the reflection process. In this process two important steps are developed.

1. In the first step, all parent child relations are investigated and conveyed to the XML file. For example, if a form contains a menu, all menu items of this menu are considered as children for the main menu component. This is true for all types of containers such as: Menus, Forms, Panels, Group Boxes, Trees, DataGrids, etc. This is parsed down the containers until reaching leaf controls.
2. In the second step and in order to connect all forms with each other, the main entry form is considered a high level component. Later one, all forms that are called or reached from the main form are considered as child forms of the main form. This is repeated for all lower level forms accordingly. By default, the parent form is not a parent for other controls. However, we assumed that by convention to connect all application forms together.

The second action that is developed in the GUI state application is the (Check) method that compares the original XML file with the currently generated one from the user interface. Parsing the current state of the user interface occurred dynamically and performs the same steps mentioned earlier in the (Add) method. Moreover, the check method will make an XML file comparison process in which the two XML files are compared node by node and attribute by attribute. This current implementation of XML files' comparison does not consider the fact that XML files maybe developed with different standards. XML has different standards. It is not uncommon to see different XML-based standards that specify elements named Name and item, for instance. Each of these element names can have a completely different meaning, depending on the standard. In one standard, the Name element might be a customer name; in another standard, it might be a product name.

The process compares the two XML files to check whether any one of the 3 possible state changes described earlier exist in comparison between the two files. The third action (Update) simply modifies the original preserved GUI state to a new one. This is needed if we have an original GUI state used for comparison and where we need to change or update that original state. The comparison process uses the basics of files or strings comparison where the two XML files are parsed and read line by line or node by node looking for differences.

In order to evaluate the value of the developed algorithm and tool developed in this research, several open source applications are used. In the first step, we parse all original GUI states of those applications to XML files. Later on, we injected many modifications in the user interfaces. In each time, the modified user interface is serialized again. The two XML files (i.e. the original and the modified ones) are compared and the developed XML files' comparison algorithm was able to detect the majority of those changes. The changes are developed to consider all six parameters described above. Through XML file comparison, developed algorithms were able to discover all changes that were injected on the tested GUIs.

GUI States reduction

In GUI, the state is defined based on GUI components, their attributes and values. Each GUI has many forms, each form has many controls, and each control has many properties. The general definition used that a change in any control property causes a GUI state change means that the number of states any GUI can have, is very large even for a small application.

However, the GUI is hierarchical by default, so is XML. Serializing those components to an XML file, rather than a database for example is a contributor to GUI states reduction. Even though, the possible states to generate can still be large. The automatic comparison and checking of the overall GUI state (i.e. through XML files' comparison) is not intended to be used for test case generation. It can be used to trigger the execution of regression testing in the same way a sensor triggers switching on, or off, an air condition system.

When using XML to store the GUI tree, we introduced a new definition for a GUI state. Rather than assuming that the GUI state depends on each property for each control in the whole application, we define the GUI state as the hierarchy that is embedded in the XML tree. A GUI state change here means a change in any parent-child relation, or any change in the specific properties that are parsed.

This definition produces an effective reduction in GUI states. For example a small application like Notepad, can have more than 200 controls, each of which may have more than 40 properties, this produces 200 * 40 =8000 potential states. Any property or control from the 8000 controls and properties triggers the state change. In our assumption, 200 controls will have 200 parent-child relations (or less). Each control has less than 10 relevant parsed properties. The total possible GUI states are reduced to 2000 or a 75% total reduction.

There is also state reduction from selecting specific properties highly relevant to the GUI model (rather than parsing all properties). Even with those assumptions, a small application can still have a large number of possible states. Reducing the possible states will have several benefits such as reducing the number of possible test cases required to achieve an acceptable coverage.

The other issue that causes another state reduction is the hierarchy. The above number is assuming that all events can occur exclusively. By enforcing the rules of the GUI hierarchy in the automatic generation of test cases, we can prevent many states from being reached from other states. For example, in Notepad, in order to reach the control “Save”, the “File” control should be selected first. As explained earlier, we can also get states reduction by abstracting the processes. This is related to the idea of build event templates for selected events. There are many GUI events, such as saving text to a file, opening a file, copying text, printing a file, changing a control color, renaming a control, .etc. that have common aspects. Such events can be defined in a library that includes the event, its pre and post conditions and the expected results if that event is successful. In such scenario, abstraction is used to ignore some details of the event that is considered specific to that event. Whenever we want to describe similar events in the same manner, we have to exclude some parts that do not apply to all those events.

4. CONCLUSION AND FUTURE WORK

The idea of defining the GUI state as the collection state for all its controls, such that a change of a single property in one control leads to a new state is valid, but is the reason for producing the huge number of possible GUI states. In software testing, we need to prioritize testing and retest the states or conditions that are critical over trying to exhaustingly test all possible GUI states. In GUI testing, we usually use a combination of users and test automation to provide the best testing adequacy or coverage. For GUI test automation in general and regression testing in particular, we are interested to reevaluate and re-execute the test suite in some particular cases and not in every GUI state change.

The automatic comparison and verification of the overall GUI state (i.e. GUI structure) is not intended to be used for test case generation. It can be also used to in regression.

We considered only the standard XML format. In future, we will develop an adaptor to deal with the different XML formats.

There are many reasons and justifications for the need to be able to dynamically know if a user interface state is changed. Having to do this manually is very complex and time consuming. In many scenarios, this can be implemented as a boot up or a built-in test where this comparison will be launched at startup.

REFERENCES:

[1] Memon A. M., 'A comprehensive framework for testing graphical user interfaces', Ph.D. thesis. Department of computer science, university of Pittsburgh (2001).

[2] Xie Q., 'Developing cost-effective model-based techniques for GUI testing', In proceedings of the 28th international conference of software engineering (ICSE’06). Shanghai, China. (2006) 997–1000.

[3] Memon A. M., and Xie Q., 'Studying the fault detection effectiveness of GUI test cases for rapidly evolving software', IEEE transactions on software engineering. Volume 31, Issue 10 (2005) 884–896.

[4] Memon A., Banerjee I., Nagarajan A., 'GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing', Proceedings of the 10th Working Conference on Reverse Engineering, (WCRE’03), (2003) 260.

[5] Memon A. M., 'Developing testing techniques for event-driven pervasive computing applications', In: OOPSLA'04 Workshop on Building Software for Pervasive Computing (BSPC'04), Vancouver, Canada; 2004.

[6] Memon A.M., 'GUI Testing: Pitfalls and Process', IEEE Computer, (2002) 87-88.

[7] Memon A. M., Pollack M. E., Soffa M. L., 'Hierarchical GUI Test Case Generation Using Automated Planning', IEEE Transactions on Software Engineering, Vol 27, number 2. (2001) 144-155.

[8] Karam M., Dascalu S., and Hazime R.H., 'Challenges and opportunities for improving code-based testing of graphical user interfaces', Journal of Computational Methods in Sciences and Engineering, IOS Press, The Netherlands, 6 (5-6), supplement 2: (2006) 379-388.

[9] Memon A. M., Banerjee I., and Nagarajan A., 'What Test Oracle Should I Use for Effective GUI Testing? ', In Proceedings of 18th IEEE International Conference on Automated Software Engineering. 2003 164–173.

[10] Tung R. and Tong K., 'A Multi-Mission Deep Space Telecommunications Analysis Tool: The Telecom Forecaster Predictor', IEEE Aerospace 2000, Big Sky, MT, 2000.

[11] Cheung Kar-Ming, Tung R. H., Lee C. H., 'Development Roadmap of an Evolvable and Extensible Multi-Mission Telecom Planning and Analysis Framework', California Institute of Technology,<http://trsnew.jpl.nasa.gov/dspace/bitstream-/2014/40407/1/03-1405.pdf> (2008).

[12] Alsmadi I. and Magel K., 'GUI Path Oriented Test Generation Algorithms'. In Proceeding of IASTED (569) Human-Computer Interaction (2007).

[13] Alsmadi I. and Magel K., 'An Object Oriented Framework for User Interface Test Automation'. MICS07 (2007).

[14] Richard Hesketh (1992). The Dirt User Interface Builder. Technical report, Computing Laboratory, University of Kent, Canterbury.

[15] Glade: A user interface designer, the glade project, http://glade.gnome.org , 2009.

[16] Andrew Krause, Foundations of GTK+ Development (Expert's Voice in Open Source), Apress; 1st ed. 2007. Corr. 2nd printing edition (April 23, 2007).

[17] Siyou Qian, Fan Jiang, An Event Interaction Structure for GUI Test Case Generation, 2nd IEEE International Conference on Computer Science and Information Technology (ICCSIT 2009).

[18] Shoichi Yokoyama, Haruhiko Sato, and Masahito Kurihara, User-Friendly GUI in Software Model Checking, Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics. 2009.

[19] Svetoslav Ganov, Sarfraz Khurshid, Dewayne Perry, A Case for GUI Testing Using Symbolic Execution, Poster Abstract, Testing: Academia and Industry Conference - Practice And Research Techniques, 2007.

[20] Xun Yuan, and Atif Memon, Alternating GUI Test Generation and Execution, Testing: Academia and Industry Conference - Practice And Research Techniques, 2008.