2011

Activities and Trends in Testing Graphical User Interfaces Automatically

Izzat M. Alsmadi
Texas A&M University-San Antonio, ialsmadi@tamusa.edu

Journal of Software Engineering

ISSN 1819-4311

Academic Journals Inc.

www.academicjournals.com

Journal of Software Engineering 5 (1): 1-19, 2011
ISSN 1819-4311 / DOI: 10.3923/jse.2011.1.19

Activities and Trends in Testing Graphical User Interfaces Automatically

Izzat Alsmadi
Department of Information Technology, Faculty of Computer and Information Technology, Yarmouk University, Irbid, Jordan

ABSTRACT

This study introduced some new approaches for software test automation in general and testing graphical user interfaces in particular. The study presented ideas in the different stages of the test automation framework. Test automation framework main activities include test case generation, execution and verification. Other umbrella activities include modeling, critical paths selection and some others. In modeling, a methodology is presented to transform the user interface of applications into XML (i.e., eXtensible Markup Language) files. The purpose of this intermediate transformation is to enable producing test automation components in a format that is easier to deal with (in terms of testing). Test cases are generated from this model, executed and verified on the actual implementation. The transformation of products' Graphical User Interface (GUI) into XML files also enables the documentation and storage of the interface description. There are several cases where we need to have a stored documented format of the GUI. Having it in XML universal format, allows it to be retrieved and reused in other places. XML Files in their hierarchical structure make it possible and easy to preserve the hierarchical structure of the user interface. Several GUI structural metrics are also introduced to evaluate the user interface from testing perspectives. Those metrics can be collected automatically using the developed tool with no need for user intervention.

Key words: Software testing, GUI, test automation, test case reduction, modeling, test case generation, test case execution

INTRODUCTION

Software testing tries to ensure that software products are not developed heuristically and optimistically. This software engineering stage ensures that the developed software is error free (This is possibly true in theory. In reality, no process can guarantee that the developed software is error free). However, this stage takes a large percent of the overall project resources. Test automation is expected to cost more at the beginning while save more eventually. There are extra costs for test automation framework at start-up. However, once correctly implemented, it is widely accepted that test automation will decrease the required number of human testers and hence reduce the cost of testing. There are many repetitive tasks in testing that take a long time and that occur very often (such as regression testing). For such activities, test automation is the best choice.

GUI test automation is a major challenge for test automation activities. Most of the current GUI test automation tools are partially automated and require the involvement of users or testers in several stages. Examples of some of the challenges that face GUI test automation is dealing with continuously new controls or components, the dynamic behavior of the user interface, timing and synchronization problems, etc.

Test automation tools are still complex and expensive. They don't fully replace testers. They can be usually used to re-execute those repeated tasks. Companies consider taking the decision to buy, or not, a GUI test automation tool as a tough decision since they don't know how much time it will require upfront for setup. They also don't know how much of test automation tasks can be fully or partially automated. We all agree that complete coverage in test automation is impractical as we also agree that complete coverage in manual testing is impractical too.

A tool is built in C# that uses reflection to serialize the GUI control components (Alsmadi and Magel, 2007; Alsmadi, 2008). The tool collects GUI controls from the application executable and then generates the XML file that contains all GUI components and their properties. After test cases are generated, they are automatically executed on the actual application. Certain control properties are selected to be serialized. These properties are relevant to the user interface testing. The application then uses the XML file that is created to build the GUI tree or the event flow graph. Test cases are then generated from the GUI XML file. Generating the test cases takes into consideration the tree structure to select the test cases that cover unique branches. Compared to other commercial and research test automation tools, this tool is considered easy to use. All activities formats are generic and universal (XML, or text files).

In test execution and verification, test cases are executed on the application and compared with the original test suite (i.e., compare the input to the test execution to its output. The input for test execution is the output of the test case generation). The advantages of this object oriented approach (i.e., test automation using the data model adapted in this research or dealing with GUI controls as objects) over the widely used capture/replay one, is in the fact that the model is generated at run time which makes it a real representative of the current state of the GUI model. In record/play back cases, we have to retest the application in case of any change in the functionalities or the GUI of the program. Once utilized, this object oriented approach is expected to be less expensive as it does not require users to manually test the application or make decision for pre- and post-conditions.

The main limitation in this approach in that it is capable of dealing with managed code. Managed code is code developed by recent object oriented languages such as C# and Java where the application contains information about its structure embedded within the application. Another limitation is dealing with synchronization and timing issues for the GUI controls. The application uses the static structure of the application as an input to test case generation. However, in some interactive applications there can be several differences between the static form and the dynamic (i.e., run time) form of the application.

There are several GUI test automation tools available in the market. Some of the larger vendors for such tools include: IBM Rational, Mercury Segue. Trying to build a GUI test automation tool is like trying to make a new operating system in that the available resources for the existing ones are hard to compete with. However, GUI test automation has not reached a mature level were those tools can be implemented in all scenarios with no problems. User Interfaces evolves rapidly with richer and newer components. The automation processes in general use artificial intelligence algorithms to simulate user activities. The richness and complexity of this space make it possible for new research ideas to compete with long time existed test automation tools in the industry.

Several researches have been done about GUI testing using the data model (Ames and Jie, 2004; Nistorica, 2005; Memon and Soffa, 2003; Memon, 2004; Memon and Xie, 2005; Sprenkle *et al*., 2005). The overall goals and approach for this work is very similar to their goals. The GUI testing framework described, as a GUI test automation structure, is generic and should be applied to any testing or GUI testing model. Since research projects in general have

limited resources, each paper discusses a specific area of GUI test automation activities. Without having all components, it is very hard to use such ideas in the industry. There is a need for using universal standards for the output formats in all testing activities. If each tool is producing several intermediate outputs (generated test cases, recorded scripts, log Files, etc.) it will be impossible to use those tools or ideas in other research projects. Some papers follow a complex procedure in test case generation and do not consider any state reductions. Assuming that changing any property in any GUI object changes the GUI state is an assumption that generated a very large number of possible states (i.e., state explosion) for even small applications. State reduction techniques are considered here to improve the effectiveness of the proposed track. There are also several studies that discuss techniques to improve test case generation with the goal of improving test coverage (Memon, 2001; Memon, 2002; Memon *et al.*, 2003; Godase, 2005; Sampath, 2006; Xie, 2006; Makedonov, 2005; Memon, 2008).

The second category of related research belongs to semi test automation using some capture/reply tools such as WinRunner, QuickTest pro, Segui silk, QARun, Rational Robot, JFCUnit, Abbot and Pounder to create unit tests for the Application Under Test (AUT). Capture/reply tools exist and have been used in the industry for years. This may make them currently, more reliable and practical as they have been tested and improved through several generations and improvements. However, there are several problems and issues in using record/play back tools. The need to reapply all test cases when the GUI changes, the complexity in editing the script code and the lack of error handlings are examples of those issues. The reuse of test oracles is not very useful in the case of using a capture/replay tool. We expect future software projects to be more GUI complex which will make the test automation data model more practical. Many researches and improvements need to be done for the suggested data model to be more practical and usable.

Several researches are presented to suggest or implement software test automation. For example, some projects use planning from AI for software testing. After analyzing the application user interface to determine what operations are possible, they (i.e., the operations) become the operators in the planning problem. Next an initial and a goal state are determined for each case. Planning is used determine a path from the initial state to the goal state. This path becomes the test plan. In our approach, test cases are generated without any user involvement (to determine the next state in the above approach which makes it semi automatic) through built in algorithms.

GUI testing framework described, as a GUI test automation structure, is generic and should be applied to any testing or GUI testing model. Since research projects in general have limited resources, each paper discusses a specific area of GUI test automation. Without having all components, it is very hard to use such ideas in the industry. There is a need for using universal standards for the output formats in all testing activities. If each tool is producing several intermediate outputs (generated test cases, recorded scripts, log files, etc) it will be impossible to use those tools or ideas in other research projects. Some studies follow a complex procedure in test case generation and do not consider any state reductions. Assuming that changing any property in any GUI object changes the GUI state is an assumption that generated a very large number of possible states for even small applications. State reduction techniques are considered here to improve the effectiveness of the track. We intended to follow the same GUI testing framework for our future work and expect the overall results to be more practical and easier to apply on actual projects.

Mustafa *et al.* (2007) focused on testing web based software. The study proposed using economic feasibility based web testing as an alternative for model based testing due to the limitation of

applying model based testing on the web. However, the challenge is in the automation of such approach in order to reduce its overall expenses.

In Sengupta (2010) study is similar to proposed study which used XML schema for GUI components regression testing. An application is developed to compare new and old test cases to differentiate between those similar and those which are new or different.

Hanna and Abu Ali (2011) studied web services robustness testing based on the platform. Besides robustness, there are several major criteria that are the focus web services testing. This includes performance, reliability and functionality.

Relative to GUI testing, Mao *et al.* (2006) focused on GUI testing and selection based on user sessions (statistically) or usage to model the usage of the software. Usage paths are collected using Windows Navigation Networks (WNNs) based on transition probability. Those can be also collected automatically from the usage log.

Xin *et al.* (2010) study mixed the using of user profile and Marcov chain for automatically deriving the testing model to improve the reliability of the software. Reliability testing is increasingly important for critical systems, particularly synchronous or reactive systems where even minor errors can be risky and expensive.

TEST AUTOMATION FRAMEWORK

The purpose of the GUI modeler is to transform the user interface to a form or model that is easier to test using an automated tool. Figure 1 represents a generic software testing framework (Makedonov, 2005). Present study translated the GUI implementation into a model that is easier for test automation processes.

Some abstraction is used in removing those properties that are less relevant or important to the GUI state to reduce the large number of possible states. The manually selected properties are: the control name, parent name, text, locationX (i.e., horizontal location), locationY (i.e., vertical location), text, or caption, forecolor, backcolor, enabled and visible.

In the process of developing test generation techniques, several test generation algorithms are developed. All algorithms check for a valid selection of a tree edge. For example, using Notepad as the AUT (A research application developed particularly for testing purposes and has capabilities similar to MS Notepad) if, for example, the current control is File, then a test algorithm may select

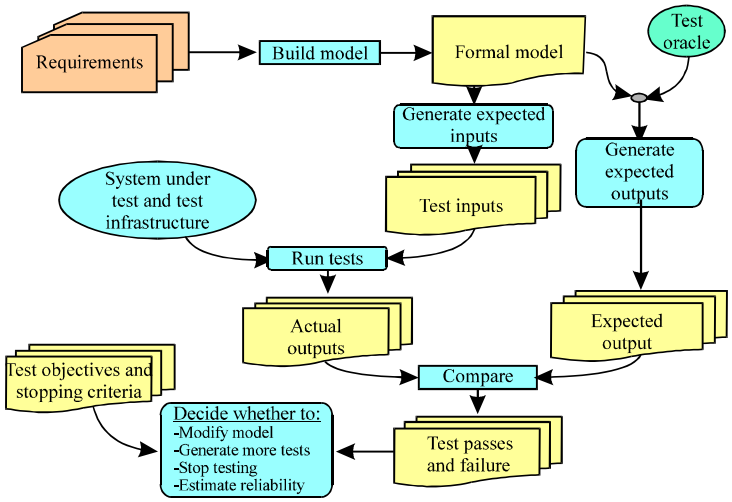

Fig. 1: Generic software testing framework

```
1,NOTEPADMAIN,SAVE,SAVEFILEBUTTON1,,,
2,NOTEPADMAIN,EDIT,FIND,TABCONTROL1.TABFIND,
3,NOTEPADMAIN,VIEW,STATUS BAR,,,
4,NOTEPADMAIN,FIND,TABCONTROL1.TABREPLACE,REPLACETABTXTREPLACE,
5,NOTEPADMAIN,FIND,TABCONTROL1,TABREPLACE,REPLACETABLABEL2,
6,NOTEPADMAIN,EDIT,CUT,,,
7,NOTEPADMAIN,EDIT,FIND,TABCONTROL1,TABREPLACE,
8,NOTEPADMAIN,OPEN,OPENFILECOMBOBOX4
```

Fig. 2: A snapshot output sample from a test case generation algorithm

randomly a valid next control from the children of the File control (e.g., Save, SaveAs, Open, Exit, Close, Print). In another algorithm, we processed the selected test scenarios to ensure that no test scenario will be selected twice in a test suite. Figure 2 is a sample output generated from one of the test generation algorithms.

In the algorithm, each test case starts from the root or the main entry Notepad Main and then selects two or more controls randomly from the tree. The algorithm verifies that the current test case has not already been generated in the already generated test cases.

To evaluate test generation efficiency in the generated test cases, the total number of arcs visited in the generated test cases is calculated to the total number of arcs or edges in the AUT. This is assuming that the AUT is transformed to a graph or tree model that represents the structure of the code. Graphs are the most commonly used structure for testing. File- Save, Edit-Copy, Format-Font are examples of arcs or edges. An algorithm is developed to count the total number of edges in the AUT by using the parent info for each control (This is a simple approach of calculating test efficiency. More rigorous efficiency measuring techniques are planned in future work). Of those tested applications, about 95% of the application paths can be tested using 200-300 test cases.

GUI modeling: The tool analyses the GUI and extract its hierarchical tree of controls or objects. The decision to use XML File as a saving location is in the fact that XML Files support hierarchy. This hierarchy can be persisted from the XML File. We encoded a level code for each control. The Main Window is considered of level 0 and so on. A Notepad version is designed as a managed code in C# to be used for testing. For this version the total number of controls for the whole application came up to 1133. There may be some redundancy in this list as roots in each file are listed twice to preserve the hierarchy.

Following is the over all counts of the controls in the different levels that is generated from the Notepad application.

<Total-Count>
<Total-Application controls>1133</Total-Application controls>
<level 0-controls>55</level 0-controls>522</level 1-controls>
<level 2-controls>310</level 2-controls>level 3-controls>246</level 3-controls></Total-Count>

For test adequacy, each path of the GUI tree should be tested or listed at least once in the test cases. A complete test case is a case that starts from level 0 and select an object from level 1 and so

on. This is with taking the hierarchical structure into consideration and selecting for example an object from level 2 that is within the reach of the selected object in level 1.

A partial test case can take two objects. Test case selection should also take in consideration the number of controls or objects in each level. For the above example, level 0 will take 55/1133 = 4.85% of the total test cases, level 1 will take 522/1133 = 46.07% of the total, level2 will take 41/183= 27.36% and level 3 will take 246/1133 = 21.71% of the total.

We should decide whether we want to run a full test case which means a test case that start from level0 and go back to level0 (in the graph path, start from top, traverse to an end and then go back to the top), or partial test cases. The test case can take the full route from level 0 to the deepest level and then back on the same path (or another) to the same original point.

The limitation we should consider is that controls in levels 1 and up may not be accessible directly or randomly and they have to be accessed through their level0 control. Also since we are pulling all controls together, the naming convention should be unique so that the object can be known, or otherwise we should use its full name.

To preserve the GUI hierarchical structure from the XML File, the application parses it to a tree view. Two different terminologies (encoded in the tree) were used:

- **Control-level:** This is the vertical level of the GUI control . The main GUI is considered the highest level control or control 0, then we keep adding 1 as we go down the hierarchy
- **Control-unit:** This is the horizontal level of the control. For example, in Notepad, File menu, with all its sub unit are unit 1, Edit menu unit 2, Format, unit 3 and so on

Generally, each full test case has to start from control-level zero and go back to this point. As an example, a test case will not be considered a valid test case if it moves from File to Edit to Format (the main menus). The test case has to move at least one level up or down. Each control is defined by where it is located horizontally and vertically. Checking for consistency in the graph should verify that there are no two objects that have the same control unit and level.

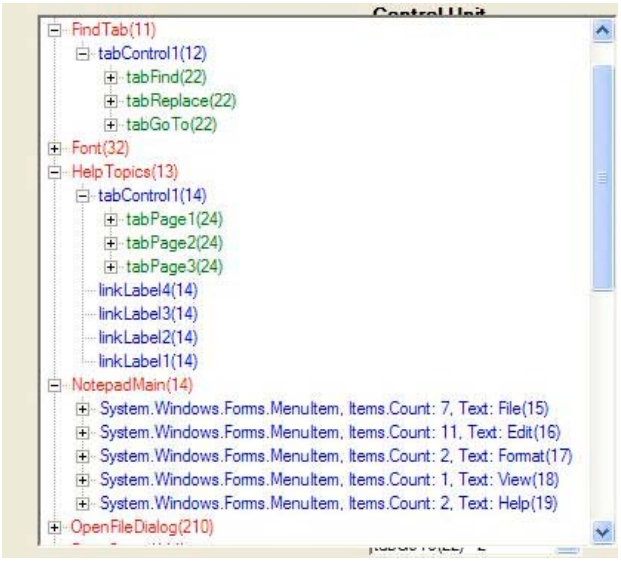

Fig. 3: Noteped control units and levels

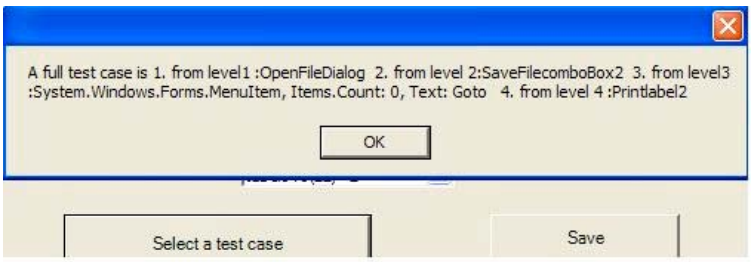

Fig. 4: A full test case generated

Figure 3 shows Notepad with the control unit and level for its GUI objects. This is a graph of the GUI objects showing the control level and unit for each object.

Getting the event flow graph from the above tree is straightforward. A particular event will occur in the tree if moving within the same control unit number one control level down. For example, File (15)-Save (13) is a valid event since both have the same control unit number "1". We can then pick the number of test cases to generate and the tool will generate full or partial test cases that observe the previous rules. Figure 4 shows an example of a full test case that the tool generates. Numbers in the above GUI components (i.e., File (15), Save (13) represent encoding values that uniquely locate each control in the GUI graph.

Present study used some abstraction removing those properties that are less relevant to the GUI state and testing to reduce the large number of possible states. In order to build a GUI test oracle, we have to study the effect of each single event. The test case result will be the combining results of all its individual events effects. For example, if we have a test case as File-Save-Edit- copy- select test-paste, then the result of this test case has effects on two objects; File-Save has effect on a File. We should study the File state change and not the whole GUI state change. Then Edit-Copy has an effect on the clipboard object, Paste will have the effect on the main editor object or state, adding the copied text.

Verifying the results of this test case will be by verifying the state change of the three objects affected; The File, the clipboard and the object editor. For such scenarios, we may need to ignore some intermediate events. Each application should have a table like Table 1, to be checked for test oracles.

Test case generation: In test case generation, several Artificial Intelligent (AI) heuristically based algorithms are created for the automatic generation of test cases. We proposed and implemented those algorithms to ensure a certain degree or level of coverage in the generated test cases where such algorithms try to verify that generated test cases are not repeated or redundant. Those algorithms are heuristic as they are not based on mathematical formula or background. Those algorithms manually verify that randomly generated test cases are new or they will be eliminated. In one algorithm, all GUI paths are given an initial weight for test case selection and those weights are reduced every time an edge or path is selected. The following dynamically-created test generation algorithms are heuristics. The goal is to generate unique test cases that represent legal test scenarios in the GUI tree with the best branch coverage. This is guaranteed by creating a unique test scenario each time. Here is a list of some of the developed algorithms.

Random legal sequences: In this algorithm, the tool first randomly selects a first-level control. It then picks a child for the selected control and so on. For example, in a Notepad AUT, If Notepad

Table 1: A sample table for GUI events effects using notepad menus

Control-event pair	Control originating the event	The object(s) affected	Effect
File, save	File-save	A File	The text from the object editor will be saved to the specified file
Edit, cut	Edit-cut	Clipboard, object editor	Moving a text-image to clipboard, clearing the original location
Edit, copy	Edit-copy	Clipboard	Moving a text-image to clipboard, keep a copy in the original location.
Edit, paste	Edit-paste	Object editor	Copying a text/image to a destination

```
1,NOTEPADMAIN,OPEN,OPENFILELABEL2,,,
3,NOTEPADMAIN,VIEW,STATUS BAR,,,
5,NOTEPADMAIN,HELPTOPICSFORM,HELPTOPICS,INDEX,LABEL4,
7,NOTEPADMAIN,TXTBODY,,,,
9,NOTEPADMAIN,TXTBODY,HELPTOPICS,CONTENT,LABEL1,
11,NOTEPADMAIN,HELPTOPICSFORM,LINKLABEL4,,,
13,NOTEPADMAIN,TXTBODY,LINKLABEL4,,,
15,NOTEPADMAIN,FILE,EXIT,,,
17,NOTEPADMAIN,EDIT,DELETE,,,
19,NOTEPADMAIN,HELP,ABOUT NOTEPAD,,,
21,NOTEPADMAIN,PRINTER,PRINTERBUTTON1,,,
23,NOTEPADMAIN,EDIT,REPLACE,,,
25,NOTEPADMAIN,FONT,FONTLISTBOX1,,,
27,NOTEPADMAIN,FIND,TABCONTROL1,TABREPLACE,REPLACETABLABEL3,
29,NOTEPADMAIN,SAVEAS,SAVEFILELABEL6,,,
31,NOTEPADMAIN,TXTBODY,ABOUT NOTEPAD,,,
33,NOTEPADMAIN,HELP,HELP TOPICS,,,
35,NOTEPADMAIN,TXTBODY,HELP TOPICS,,,
37,NOTEPADMAIN,SAVE,SAVEFILELABEL5,,,
39,NOTEPADMAIN,FORMAT,FONT,FONTGROUPBOX1,FONTTEXTBOX4,
41,NOTEPADMAIN,SAVE,SAVEFILELISTBOX1,,,
43,NOTEPADMAIN,FORMAT,FONT,FONTLISTBOX2,,
45,NOTEPADMAIN,SAVEAS,SAVEFILELABEL2,,,
47,NOTEPADMAIN,FORMAT,WORD WRAP,,,
49,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
```

Fig. 5: A sample output from the random legal-sequence algorithm

main menu is selected as the first level control; candidate children are File, Edit, Format, View and Help. In the second step, if the File control is selected from those children, the children for File (i.e., Save, SaveAs, Exit, Close, Open, Print, etc) are the valid next-level controls from which one control is randomly selected and so on. The summary of steps for this algorithm is:

- Select dynamically (i.e., the tool) the main entry control (or select any control of level 0)
- Find all the children for the control selected in one and randomly pick one child
- Find all the children for the control selected in two and select one child control
- Repeat three until no child is found for the selected control. The test scenario for this cycle is the sequence of controls from all previous steps
- Repeat the above steps for the total number of the required test scenarios

Figure 5 is a sample output dynamically generated from the random legal sequence algorithm.

```
WHILE numtestcases < testcase_required
SELECT control_root
DISPLAY control_root
WHILE control_root has new child
DISPLAY child
END WHILE
INCREMENT numtestcases
END WHILE
```

Fig. 6a: Pseudo code for Random less controls

Random less previously selected controls: In this algorithm, controls are randomly selected as in the previous algorithm. The only difference is that if the current control is previously selected (e.g., in the test case just before this one), this control is excluded from the current selection. This causes the algorithm to always look for a different new control to pick. The pseudo code for this algorithm is as follows:

- Select the main entry control (or select any first level control)
- Find all the children for the control from one and select one child control
- Save the selected control to a variable. Check if this is the previously selected control. If this is the same control selected in the previous scenario, pick another one
- Find all the children for the previously selected control and pick one child control. Save the selected control to a variable. Check the previously selected control. If they are the same, select another one
- Repeat four until no child is found for the selected control. The test scenario of this cycle is the controls in sequence from all previous steps
- Repeat the above steps for the total number of required test scenarios

Figure 6a and b represent the pseudo code.

Excluding previously generated scenarios: Rather than excluding the previously selected control as in the second algorithm, this algorithm excludes all previously generated test cases or scenarios and hence verifies the generation of a new unique test case every time. The scenario is generated and if the test suite already contains the new generated scenario, it will be excluded and the process to generate a new scenario starts again. In this scenario, the application may stop before reaching the number of required test cases (requested by the user) to generate if there are no more unique test scenarios to create. As explained earlier, the algorithm is given limited resources. It is expected to find the solution within those resources or the algorithm stops and is considered to have failed (similar to the way ping command in networks works; given limited time to test reachability).

The steps for this algorithm are:

- Select a first level control
- Find all the children for the selected control in step one and randomly pick one child of that control
- Find all the children for the control selected in step two and pick one child of that control

```
0,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
1,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
2,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
3,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
4,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
5,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
6,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
7,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
8,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
9,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
10,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
11,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
12,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
13,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
14,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
15,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
16,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
17,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
18,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
19,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
20,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
21,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
22,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
23,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
24,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
25,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
26,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
27,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
28,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
29,NOTEPADMAIN,SAVEAS,SAVEFILELISTBOX1,,,
30,NOTEPADMAIN,PAGESETUP,PAGESETUPLABEL4,,,
```

Fig. 6b: A sample from the algorithm; random less previously selected controls

- Repeat step three until no child is found for the selected control

The test scenario for this cycle is the controls in sequence from all previous steps. Save the sequence of the test scenario to a Hashtable (e.g., a data structure)

- Check the scenarios that are existed in the Hashtable. If the current created scenario is in the Hashtable, exclude it from the selection and restart the selection process
- Repeat the above steps for the total number of the required test scenarios unless a termination process is called. Figure 7 shows a sample output from the above algorithm

As seen in the sample, some of the generated test cases are canceled as they were previously generated (from looking at the sequence of the test cases).

```
1,NOTEPADMAIN,PRINTER,PRINTERLABEL10,,,
3,NOTEPADMAIN,SAVEAS,SAVEFILELABEL4,,,
5,NOTEPADMAIN,ABOUT,ABOUTHELPLABEL2,,,
7,NOTEPADMAIN,PAGESETUP,PAGESETUPGROUPBOX1,PAGESETUPRADIOBUTTON1,,
9,NOTEPADMAIN,VIEW,STATUS BAR,,,
11,NOTEPADMAIN,FILE,EXIT,,,
13,NOTEPADMAIN,BUTTON3,STATUS BAR,,,
15,NOTEPADMAIN,ABOUT,ABOUTHELPLABEL3,,,
17,NOTEPADMAIN,BUTTON3,,,,
19,NOTEPADMAIN,BUTTON3,ABOUTHELPLABEL3,,,
21,NOTEPADMAIN,BUTTON3,ABOUTHELPLABEL2,,,
23,NOTEPADMAIN,FONT,FONTLABEL4,,,
25,NOTEPADMAIN,FORMAT,FONT,FONTLABEL4,,
27,NOTEPADMAIN,FONT,FONTLISTBOX3,,,
29,NOTEPADMAIN,HELPTOPICSFORM,HELPTOPICS,INDEX,LABEL3,
31,NOTEPADMAIN,OPEN,OPENFILELABEL8,,,
33,NOTEPADMAIN,HELPTOPICSFORM,LINKLABEL4,,,
35,NOTEPADMAIN,EDIT,UNDO,,,
37,NOTEPADMAIN,BUTTON3,UNDO,,,
39,NOTEPADMAIN,EDIT,REPLACE,,,
```

Fig. 7: A sample of the unique-scenarios algorithm

Weight selection algorithm: In this scenario, rather than giving the same probability of selection or weight for all candidate children of controls, as in all previous scenarios, in this algorithm any child that is selected in the current node causes its weight (i.e., probability of selection) next time to be reduced by a certain percent. If the same control is selected again, its weight is reduced again and so on.

The summary of steps for this algorithm is:

- Select the first level control
- Select randomly a child for the control selected in step one. Give equal weights for all children. Decrease weight for the selected one by a fixed value
- Find all the children for the control selected in step two and randomly pick one child control. Give equal weights for all children and decrement the weight for the selected one by the same fixed value (this value can be the same for all levels, or each level can have a different value).
- Repeat step three until no child is found for the selected control
- The test scenario for this cycle is the sequence of the selected controls from all the previous steps
- Repeat the above steps for the total number of the required test scenarios unless a termination process is called

Keep the decreased weights from the earlier scenarios. Figure 8 is a sample output from the weight selection algorithm.

Both algorithms; three and four, are designed to ensure branch coverage, all-paths testing (i.e., testing that experiences all possible paths of an application. Paths usually represent different decisions in the code), and reduce redundancy in the generated test suite.

```
1,NOTEPADMAIN,HELPTOPICSFORM,HELPTOPICS,INDEX,LABEL3,
3,NOTEPADMAIN,SAVEAS,SAVEFILELABEL2,,,
5,NOTEPADMAIN,EDIT,PASTE,,,
7,NOTEPADMAIN,BUTTON3,,,,
9,NOTEPADMAIN,FIND,TABCONTROL1,TABFIND,FINDTABBTNCANCEL,
11,NOTEPADMAIN,VIEW,STATUS BAR,,,
13,NOTEPADMAIN,HELP,HELP TOPICS,,,
15,NOTEPADMAIN,ABOUT,ABOUTHELPLABEL1,,,
17,NOTEPADMAIN,FORMAT,FONT,COMBOBOX1,,
19,NOTEPADMAIN,FORMAT,FONT,FONTGROUPBOX1,FONTTEXTBOX4,
21,NOTEPADMAIN,OPEN,OPENFILEBUTTON2,,,
23,NOTEPADMAIN,HELP,ABOUT NOTEPAD,,,
25,NOTEPADMAIN,SAVE,SAVEFILELABEL9,,,
27,NOTEPADMAIN,FIND,TABCONTROL1,TABFIND,FINDTABTXTFIND,
29,NOTEPADMAIN,SAVEAS,SAVEFILELABEL3,,,
31,NOTEPADMAIN,PRINTER,PRINTERLABEL9,,,
33,NOTEPADMAIN,FILE,OPEN,OPENFILECOMBOBOX1,,
35,NOTEPADMAIN,FONT,FONTGROUPBOX1,FONTTEXTBOX4,,
37,NOTEPADMAIN,SAVE,SAVEFILELABEL1,,,
39,NOTEPADMAIN,FONT,FONTLISTBOX1,,,
41,NOTEPADMAIN,EDIT,REPLACE,,,
43,NOTEPADMAIN,PAGESETUP,PAGESETUPBUTTON2,,,
45,NOTEPADMAIN,FIND,TABCONTROL1,TABREPLACE,REPLACETABCHKMATCHCASE,
47,NOTEPADMAIN,FILE,PRINT,PRINTTAB,PRINTGROUPBOX1,
49,NOTEPADMAIN,FONT,FONTLABEL2,,,
```

Fig. 8: A sample output from the weight selection algorithm

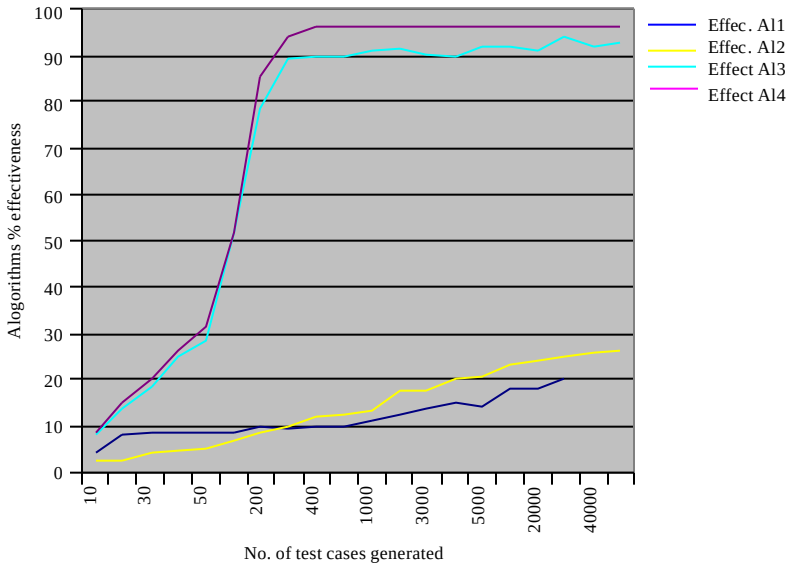

Fig. 9: Test suite effectiveness for the four algorithms explained earlier

We define test suite effectiveness that can be calculated automatically in the tool in order to evaluate the above algorithms. Test suite effectiveness is defined as the total number of edges discovered to the actual total number of edges. Figure 9 shows test effectiveness for the four algorithms explained earlier.

As shown above, the last two algorithms reach to approximately 100% effectiveness by generating less than 300 test cases.

To evaluate test generation efficiency in the generated test cases, the total number of arcs visited in the generated test cases is calculated to the total number of arcs or edges in the Application Under Test (AUT). File-Save, Edit-Copy, Format-Font are examples of arcs or edges. An algorithm is developed to count the total number of edges in the AUT by using the parent info for each control. (This is a simple approach of calculating test efficiency. More rigorous efficiency measuring techniques are planned in future work). Of those tested applications, about 95% of the application paths can be discovered and tested using 200-300 test cases.

Critical path testing and test case prioritization: Critical paths are the paths that cause test execution to be slow, or the paths that have more probability of errors than the other testing paths. Here are some of the critical path examples:

- An external API or a command line interface accessing an application
- Paths that occur in many tests (in a regression testing database)
- The most time consuming paths

Three algorithms are developed to find critical paths automatically in the AUT.

Critical paths using node weight: In this approach, each control is given a metric weight that represents the count of all its children. For example if the children of File are: Save, SaveAs, Close, Exit, Open, Page Setup, and Print, then its metric weight is seven (another alternative is to calculate all the children and grand children). For each generated scenario, the weight of that scenario is calculated as the sum of all the weights of its individual selected controls. Table 2 is a sample output generated that calculates the weight of some test scenarios for an AUT.

To achieve coverage with test reduction, the algorithm selects randomly one of those scenarios that share the same weight value as a representative for all of them. An experiment should be done to test whether those scenarios that have the same weight can be represented by one test case or not (i.e., from a coverage perspective).

Here are some observations about the results of this critical path selection using our Notepad AUT.

- The maximum scenario weight is 40. Many test scenarios have the same maximum weight
- Since the same weight usually indicates similar tree depth, we can automate the selection process (by selecting arbitrarily one of those that have equal weight). This may give us a very reasonable test cases' reduction
- Tree weight values are relative and dependent on the main entry selection
- If there is a node that does not have children, its total metric count is like its parent count as its metric count = 0

Other alternative is to set a minimum weight required to select a test scenario and then generate all test scenarios that have a weight higher than the selected cut off. The two criteria that

Table 2: Test scenarios' weights metric

Test scenario	Weight (No. of controls)
NOTEPADMAIN FILE PRINT PRINTTAB PRINTBUTTON2	28
NOTEPADMAIN FILE PRINT PRINTTAB PRINTLABEL3	28
NOTEPADMAIN FILE PRINT PRINTTAB PRINTLABEL1	28
NOTEPADMAIN FILE PRINT PRINTTAB PRINTBUTTON1	28
NOTEPADMAIN FILE PRINT PRINTTAB	28
NOTEPADMAIN FILE PRINT	28
NOTEPADMAIN PAGESETUP PAGESETUPGROUPBOX2 PAGESETUPLABEL7	34
NOTEPADMAIN PAGESETUP PAGESETUPGROUPBOX2 PAGESETUPLABEL6	34
NOTEPADMAIN PAGESETUP PAGESETUPGROUPBOX2 PAGESETUPTEXTBOX3	34
NOTEPADMAIN PAGESETUP PAGESETUPGROUPBOX2 PAGESETUPLABEL8	34
NOTEPADMAIN PAGESETUP PAGESETUPGROUPBOX2 PAGESETUPTEXTBOX1	34
NOTEPADMAIN PAGESETUP PAGESETUPGROUPBOX2 PAGESETUPTEXTBOX4	34
NOTEPADMAIN PAGESETUP PAGESETUPGROUPBOX2 PAGESETUPTEXTBOX2	34

Table 3: Weight algorithm reduction percentages

AUT	Total No. of test scenarios	Coverage reduction percentage
Notepad	174	94.25
FP analysis	28	82.14
WordNet	8	75.00
Gradient	153	92.81
GUI controls	51	88.23
Hover	10	90.00

affect the critical path weight factor are the number of nodes that the path consists of and the weight of each node. This technique can help us dynamically define the longest or deepest paths in an application. For example, all the 40 weight values in Notepad as the AUT belong to the node that has the page setup form.

Table 3 shows the reduction percentage of selected scenarios using the above selection algorithm (given the assumption that the same weight scenarios can be represented by one as explained earlier). As the tool uses reflection to generate XML files from the AUT, this tool can handle only managed code in which the application keeps information about itself. Application from C++ for example, cant be tested using this tool. The Notepad application listed in this experiment is a managed version developed to have similar functionalities of MS Notepad.

Critical path level reduction through selecting representatives: This technique approaches test selection reduction through selecting representative test scenarios. Representatives are elected from the different categories, classes or areas to best represent the whole country. In this approach, the algorithm arbitrarily selects a test scenario. The selected scenario includes controls from the different levels. Starting from the lowest level control, the algorithm excludes from selection all those controls that share the same parent with the selected control. This reduction shouldn't exceed half of the tree depth. For example if the depth of the tree is four levels, the algorithm should exclude controls from levels three and four only.

We assume that three controls are the least required for a test scenario (such as Notepad-File-Exit). We continuously select five test scenarios using the same reduction process described above. The selection of the number five for test scenarios is heuristic. The idea is to select the least amount

Table 4: Level reduction test results' sample

Test scenarios	Total percent of test cases' reduction (%)
NOTEPADMAIN, PRINTER, PRINTERBUTTON1,,,	
NOTEPADMAIN,SAVE,SAVELABEL7,,	
NOTEPADMAIN,EDIT,FIND,TABCONTROL1,TABFIND,FINDTABBTNNEXT	
NOTEPADMAIN,FILE,PRINT,PRINTTAB,PRINTLABEL7,	
NOTEPADMAIN,SAVE,SAVELABEL5	65.1
NOTEPADMAIN,FILE,PRINT,PRINTTAB,PRINTLISTBOX1	
NOTEPADMAIN,FONT,FONTLABEL2	
NOTEPADMAIN,HELPTOPICFORM,HELPTOPICS,SEARCH,BUTTON1	
NOTEPADMAIN,FONT,FONTTEXTBOX2,,	
NOTEPADMAIN, PRINTER, PRINTERBUTTON2,,,	41.67
NOTEPADMAIN,FILE,PRINT,PRINTTAB,PRINTGROUPBOX1	
NOTEPADMAIN, PAGESETUP,PRINTER,	
NOTEPADMAIN,FONT,FONTLISTBOX2,,	
NOTEPADMAIN,OPEN,OPENFILELABEL4,,	
NOTEPADMAIN,SAVEAS,SAVEFILECOMBOBOX2,	51.56

of test scenarios that can best represent the whole GUI. Table 4 is a sample output of measuring test case reduction using the above algorithm. The five selected scenarios are listed along with their total reduction.

Weight controls from user sessions: The previously described algorithms for critical paths' selection depend on statistics pulled from the implementation model. As an alternative, we can analyze several user captured sessions (e.g. from testers or users in beta testing) to automatically weight the GUI controls. User session data is the set of user actions performed on the AUT from entering the application until leaving it. We can analyze several user captured sessions (e.g., from testers or users in beta testing) to automatically weight the GUI controls or widgets. User session data is the set of user actions performed on the Application Under Test (AUT) from entering the application until leaving it.

We can classify a control, or a pair of controls, according to the number of times they are repeated in a user session. User sessions are likely to detect faults in the application that are not predictable in earlier testing phases. Another advantage of testing with user sessions is that testing is possible in the absence of specifications or in the presence of incorrect and incomplete specifications, which often occurs in software development.

The session logs all the controls that are executed in the different scenarios. A simple count or percentage is given to each control depending on how many times it is listed in those scenarios. The test scenarios should include all primary and major use cases for the AUT. The controls' weights (calculated from user sessions) can drive the test case generation and execution. Theoretically all controls should get the same weight in the generated test suite. However, in real scenarios this may not be true. We can use the weighing method for single controls or for a sequence of controls (result from a specific use case).

We may cluster the controls, or sequence of controls, according to their usage from user sessions into three levels; heavily used, medium and low. Depending on the availability of the resources for testing, we may choose one or two categories and generate test cases that cover those controls in the categories with a proportion to their weight or occurrence. The developed algorithm in this research is considered as a hybrid technique that uses some of the capture/ reply processes. In a capture/ reply tool, the same user session that is captured in the manual testing is executed. In this

```
1, NOTEPADMAIN, FILE,NEW,TXTBODY,,
2, NOTEPADMAIN,FILE,SAVE AS,SAVEFILEBUTTON1,,,
```

Fig. 10: A sample of generated test cases (generic)

```
Control Event Date Time
File new Menu Click 10/3/2008 11:51:23 AM File new Mouse Down 10/3/2008 11:51:23
AM File new Mouse Up 10/3/2008 11:51:23 AM
New txtbody Menu Click 10/3/2008 11:51:23 AM New txtbody Mouse Down 10/3/2008
11:51:23 AM New txtbody Mouse Up 10/3/2008 11:51:23 AM TxtBody Mouse Move
10/3/2008 11:51:23 AM TxtBody Key Down 10/3/2008 11:51:23 AM TxtBody Key Up
10/3/2008 11:51:23 AM
(Test) is written in the document 10/3/2008 11:51:23 AM (Test) is written in the document
10/3/2008 11:51:24 AM
SaveFilebutton1 Mouse Move 10/3/2008 11:51:24 AM SaveFilebutton1 Mouse Button Down
10/3/2008 11:51:24 AM SaveFilebutton1 Mouse Button Up 10/3/2008 11:51:24 AM File
SAVE AS Menu Click 10/3/2008 11:51:24 AM
File SAVE AS Mouse Down 10/3/2008 11:51:24 AM
File SAVE AS Mouse Up 10/3/2008 11:51:24 AM SaveFilebutton1 Mouse Move 10/3/2008
11:51:24 AM SaveFilebutton1 Mouse Button Down 10/3/2008 11:51:24 AM SaveFilebutton1
Mouse Button Up 10/3/2008 11:51:24 AM
```

Fig. 11: Log file output of a sample test case

approach the controls' weights are extracted from the manual testing to guide test case generation and execution. The reason for considering this track rather than using capture/ reply test execution and validation is to avoid the dependency on the absolute location of the screen and controls that is required by capture/replay tools. Having a hybrid solution may give us the best of both and utilize the accumulative experience and knowledge in different technologies.

In order to record user events, we implemented in our C# application Alsmadi and Magel, (2007), the interface IMessageFilter that is used to capture messages between Window applications and components. In the AUT, each GUI control that is triggered by the user is logged to a file that represents the user sessions. The minimum information required is the control, its parent and the type of event. The user session file includes the controls triggered by the user in the same sequence. Such information is an abstract of the user session sequence. In many cases, the same control is repeated several time (due to the nature of logging the window messages), The implementation will get rid of all those controls repeated right after each other. The same information can be extracted from the events written to the event log.

Test case execution and verification: For test verification, a log file is created to track the events that are executed in the tool during the execution process. In a simple example, Fig. 10 shown below, two test cases are generated that write a text in Notepad and save it to a file. Those test cases are generated using the tool.

The first test case opens a new document and writes to it. As part of the default input values, we set for each control a default value to be inserted by the tool through execution. A textbox writes the word "test" or the number "0" whenever it is successfully called. A menu item is clicked, using its parent, whenever it is successfully called. For example, if Save is called as a control, File-Save as an event is triggered. We should have tables for valid and invalid inputs for each GUI control. The second test case opens the save File dialogue and clicks the OK or accept button (SaveFilebutton1), to save the document. Here is the corresponding log file output for the above test cases (Fig. 11).

Since the test execution process is subjected to several environment factors, the verification process is divided into three levels.

- In the first level the tool checks that every control in the test suite is successfully executed. This step is also divided into two parts. The first part is checking that all controls executed are existed in the test suite. This is to verify that the execution process itself does not cause any extra errors. The second part that ensures all controls in the test suites are executed tests the execution and its results. In the implementation of this level, some controls from the test scenarios were not executed. This is maybe the case of some dynamic execution or time synchronization issues where some controls are not available the time they are expected
- In the second level the tool checks that the number of controls matches between the two suites
- In the third level, the tool checks that the events are in the same sequence in both suites. The verification process is automated by comparing the test cases' file with the log file. Time stamp is important to verify the correct sequence of events. The controls in the test case suites are written to a sorted list and the execution sequence is also written to another sorted lists. To verify the results of the test execution, the two lists are compared with each other. Upon testing several applications, a small percent of controls generated in the test cases and not executed. Timing synchronization causes some controls to be unavailable or invisible within their execution time. Regular users "wait" for a new menu to be opened. This time varies depending on the application, the computer and the environment. Time synchronization and some other dynamic issues are part of the future research goals

One approach for the GUI test oracle is to have event templates. For each test scenario listed, expected results are written in a way that can be automatically verified. This requires some level of abstraction where similar events are abstract into one event (like the saving process). This proposal does not mean exhaustively testing all probable events. By selecting critical events for automation and abstracting similar events we will get a valuable state reduction that makes the number of GUI states more practical in terms of testing.

CONCLUSION

Developing a user interface test automation tool faces several challenges. Some of those challenges are: Serializing the GUI widgets, test results' verification, time synchronization issues, handling dialog boxes, testing data dependent systems, and building error logging and recovery procedures. Some techniques, in test case generation, execution and verification, are explained in principles in this article. Test case generation from user sessions is explored that represent real usage of the application and focuses on the application areas where they can be heavily used by users. A logging procedure is implemented to compare the executed suite with the generated one. A formal verification procedure is presented that compare and verifies the output of the execution process with its input. Another track of verification is suggested. This track requires building templates for events. For each event pre conditions, post conditions and expected results are included. More elaborations and verifications are required to prove the effectiveness of the suggested approaches. Automation of the first few test cases is expensive; beyond that they become much cheaper to develop and execute.

Test automation can only be successful when we keep in mind that testing in general and particularly automated testing, is easily made obsolete by some changes in the application and

environment. In GUI, it is difficult to reach a high level of test adequacy in generating test cases that cover all possible combinations. GUI state reduction is needed to make testing all or most possible states feasible.

Test case generation algorithms are automatically developed to ensure test adequacy or coverage which refers to the percentage of the application that test cases cover. Test cases will be automatically generated, using a developed tool (i.e., GUI Auto) for this purpose, from the implementation transformed model. Different algorithms are produced. Their coverage is compared and tested using execution and verification. Some APIs are developed to allow the automatic execution of the generated test cases on the actual application. The tool uses reflection to recreate the code from its managed output. There are also some verification techniques produced to compare actual outputs with expected ones. Some execution techniques are developed such as a monkey testing technique, where the tool executes random sequence of events to test the robustness of the developed product. Although such sequence of events is not possible in real usage scenarios, however, good application should never crash. All those algorithms and techniques are implemented and evaluated in the developed tool. References in literature can be found to the developed tool and some of those experiments and algorithms.

In critical path testing, several algorithms such as: weighting algorithm, genetic algorithm, and representative algorithm are produced as techniques to select best representatives from test cases out of large numbers of test cases in a test oracle or database. Another trend that will be introduced is the using of user sessions in test case generation and execution. User sessions can be very useful inputs for test cases. Those scenarios can be more real and test cases are more accurate and represent actual user scenarios or usage if compared with hypothetic test cases.

In this approach, there is state reduction from selecting specific properties to parse. Those properties are more relevant and critical than the rest for the testing process. Total properties of less than 10 are selected. The idea of test automation is not to automate everything; we automate to save time and effort. The other issue that causes state reduction is considering the hierarchy. In flat state diagrams, we assume that any state is reached from another state once certain preconditions are achieved. In GUI terms it means that any control is accessible from any other one. This is not true in GUI states, as usually controls are only accessible through their parent controls. We can also get GUI states reduction by abstracting the processes. For example, the saving process has several intermediate processes (File-save-as, File name, type... ok), but what is important is the end state where there are certain results if the process is successful (e.g., a File with certain content is saved to a certain location) and other results if it is not.

We don't automate testing everything. We automate testing those frequently repeated tasks. We don't automate those tests that require user validation or those tests that require special dynamic or time synchronization consideration (such as connecting to a database or verifying the overall acceptance of a GUI).

REFERENCES

Alsmadi, I. and K. Magel, 2007. GUI path oriented test generation algorithms. Proceeding of IASTED (569) Human-Computer Interaction, March 14–16, Chamonix, France, pp: 216-219.

Alsmadi, I., 2008. The utilization of user sessions in testing. Proceedings of the 7th IEEE/ACIS International Conference on Computer and Information Science, May 14-16, Portland, OR., pp: 581-585.

Ames, A.K. and H. Jie, 2004. Critical Paths for GUI Regression Testing. University of California, Santa Cruz.

Godase, S., 2005. An introduction to software test automation. http://www.indicthreads.com/1329/an-introduction-to-software-test-automation/#.

Hanna, S. and A. Abu Ali, 2011. Platform effect on web services robustness testing. J. Applied Sci., 11: 360-366.

Makedonov, Y., 2005. Managers guide to GUI test automation. Software Test and Performance Conference.

Mao, Y., F. Boqin, H. Zhenfang and Z. Li, 2006. Important usage paths selection for GUI software testing. Inform. Technol. J., 5: 648-654.

Memon, A.M., 2001. A comprehensive framework for testing graphical user interfaces. Ph.D. Thesis, University of Pittsburgh, USA.

Memon, A.M., 2002. GUI testing: Pitfall and process. Software Technol., 35: 87-88.

Memon, A.M. and M.L. Soffa, 2003. Regression testing of GUIs. Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Sept. 1-5, Helsinki, Finland, pp: 118-127.

Memom, A., I. Banerejee and A. Nagarajan, 2003. GUI ripping: Reverse engineering of graphical user interfaces for testing. Proceedings of the 10th Working Conference on Reverse Engineering, Nov. 13-16, Computer Soiety, pp: 260-269.

Memon, A.M., 2004. Developing testing techniques for event-driven pervasive computing applications. Department of Computer Science. University of Maryland, http://www.cs.umd.edu/~atif/papers/MemonBSPC2004.pdf.

Memon, A. and Q. Xie, 2005. Studying the fault detection effectiveness of GUI test cases for rapidly evolving software. IEEE Trans. Software Eng., 31: 884-896.

Memon, A.M., 2008. Automatically repairing event sequence-based GUI test suites for regression testing. Proceedings of ACM Transactions on Software Engineering and Methodology (TOSEM), November 2008, New York, USA., pp: 1-35.

Mustafa, G., A.A. Shah, K.H. Asif and A. Ali, 2007. A strategy for testing of web based software. Inform. Technol. J., 6: 74-81.

Nistorica, G., 2005. Automated GUI testing. http://www.perl. com/pub/2005/08/11/win32guitest.html.

Sampath, S., 2006. Cost effective techniques for user session based testing of web applications. Ph.D. Thesis, University of Delaware.

Sengupta, G.J., 2010. Regression testing method based on XML schema for GUI components. J. Software Eng., 4: 137-146.

Sprenkle, S., E. Gibson, S. Sampath and L. Pollock, 2005. Automated replay and failure detection for web applications. Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, Nov. 7-11, Long Beach, USA., pp: 253-262.

Xie, Q., 2006. Developing cost-effective model-based techniques for GUI testing. Proceedings of the 28th International Conference on Software Engineering. May 20-28, Shanghai, China, pp: 997-1000.

Xin, W., H. Feng-Yan and Q. Zheng, 2010. Software reliability testing data generation approach based on a mixture model. Inform. Technol. J., 9: 1038-1043.