The Introduction of Several User Interface Structural Metrics to Make Test Automation More Effective

Izzat Alsmadi* and Mohammed Al-Kabi*

Department of Computer Information Systems, Yarmouk University, Jordan

Abstract: User interfaces have special characteristics that differentiate them from the rest of the software code. Typical software metrics that indicate its complexity and quality may not be able to distinguish a complex Graphical User Interface (GUI) or a high quality one from another that is not. This paper is about suggesting and introducing some GUI structural metrics that can be gathered dynamically using a test automation tool. Rather than measuring quality or usability, the goal of those developed metrics is to measure the GUI testability, or how much it is hard, or easy to test a particular user interface. We evaluate GUIs for several reasons such as usability and testability. In usability, users evaluate a particular user interface for how much easy, convenient, and fast it is to deal with it. In our testability evaluation, we want to automate the process of measuring the complexity of the user interface from testing perspectives. Such metrics can be used as a tool to estimate required resources to test a particular application.

Keywords: Layout complexity, GUI metrics, interface usability.

1. INTRODUCTION

The purpose of software metrics or measurements is to obtain better measurements in terms of risk management, reliability forecast, cost repression, project scheduling, and improving the overall software quality.

GUI code has its own characteristics that make using the typical software metrics such as lines of codes, cyclic complexity, and other static or dynamic metrics impractical and may not distinguish a complex GUI from others that are not. There are several different ways of evaluating a GUI that include; formal, heuristic, and manual testing. Other classifications of user evaluation techniques includes; predictive and experimental. Unlike typical software, some of those evaluation techniques may depend solely on users and may never be automated or calculated numerically. GUI structural metrics are those metrics that depend on the static architecture of the user interface. The main contribution that distinguishes this research is that it focuses on the generation of GUI metrics automatically (i.e. through a tool without user interference). In this paper, we elaborate on those earlier metrics, suggested new ones and also make some evaluation in relation to the execution and verification process. We selected some open source projects for the experiments. Most of the selected projects are relatively small, however, GUI complexity is not always in direct relation with size or other code complexity metrics.

2. RELATED WORK

Usability findings can vary widely when different evaluators study the same user interface, even if they use the same evaluation technique. Melody *et al.* surveyed 75 GUI usability evaluation methods and presented a taxonomy for comparing those various methods [1]. GUI usability evaluation typically only covers a subset of the possible actions users might take. For these reasons, usability experts often recommend using several different evaluation techniques [1].

Highly disordered or visually chaotic GUI layouts reduce usability, but too much regularity is unappealing and makes features hard to distinguish.

Some interface complexity metrics that have been reported in the literature include:

- The number of controls in an interface (e.g. Controls' Count; CC). In some applications [2], only certain controls are counted such as the front panel ones. Our tool is used to gather this metric from several programs (as it will be explained later).

- The longest sequence of distinct controls that must be employed to perform a specific task. In terms of the GUI XML tree, it is the longest or deepest path or in other words the tree depth.

- The maximum number of choices among controls with which a user can be presented at any point in using that interface. We also implemented this metrics by counting the maximum number of children a control has in the tree.

- The amount of time it takes a user to perform certain events in the GUI [3]. This may include; key strokes, mouse clicks, pointing, selecting an item from a list, and several other tasks that can be performed in a GUI. This "performance" metric may vary from a novice user on the GUI to an expert one. Automating this metric will reflect the API performance which is usually expected to be faster than a normal user. In typical applications, users don't just type keys or click the mouse. Some events require a response and think-

*Address correspondence to these authors at the Department of computer Information Systems, Yarmouk University, Jordan;
E-mails: ialsmadi@yu.edu.jo; mohammedk@yu.edu.jo

1874-107X/09

ing time, others require calculations. Time synchronization is one of the major challenges that tackle GUI test automation.

It is possible, and usually complex, to calculate GUI efficiency through its theoretical optimum [4, 5]. Each next move is associated with an information amount that is calculated given the current state, the history, and knowledge of the probabilities of all next candidate moves.

The complexity of measuring GUI metrics relies on the fact that it is coupled with metrics related to the users such as: thinking, response time, the speed of typing or moving the mouse, etc. Structural metrics can be implemented as functions that can be added to the code being developed and removed when development is accomplished [6].

Balbo [7] had a survey about GUI evaluation automation. He classified several techniques for processing log files as automatic analysis methods. Some of those listed methods are not fully automated as suggested. Many web automatic evaluation tools were developed for automatically detecting and reporting ergonomic violation (usability, accessibility, etc) and in some cases making suggestions for fixing them [7-13].

Complexity metrics are used in several ways with respect to user interfaces. One or more complexity metrics can be employed to identify how much work would be required to implement or to test that interface. It can also be used to identify code or controls that are more complex than a predefined threshold value for a function of the metrics. Metrics could be used to predict how long users might require to become comfortable with or expert at a given user interface. Metrics could be used to estimate how much work was done by the developers or what percentage of a user interface's implementation was complete. Metrics could classify an application among the categories: user-interface, processing code, editing, or input/output dominated. Finally, metrics could be used to determine what percentage of a user interface is tested by a given test suite or what percentage of an interface design is addressed by a given implementation.

In the GUI structural layout complexity metrics, there are several papers presented. Tullis studied layout complexity and demonstrated it to be useful for GUI usability [14]. However, he found that it did not help in predicting the time it takes for a user to find information [15, 16]. Tullis defined arrangement (or layout) complexity as the extent to which the arrangement of items on the screen follows a predictable visual scheme [6]. In other words, the less predictable a user interface is the more complex it is expected to be.

The majority of layout complexity papers discussed the structural complexity in terms of visual objects' size, distribution and position. Other layout structural attributes will be studied in this paper. Examples of such layouts are: GUI tree structure, the total number of controls in a GUI, and the tree layout. Our research considers few metrics from widely different categories in this complex space. Those selected user interface structural layout metrics are calculated automatically using a developed tool. Structural metrics are based on the structure and the components of the user interface. It does not include those semantic or procedural metrics that depend on user actions and judgment (that can barely be automatically calculated). These selections are meant to illustrate the richness of this space, but not to be comprehensive.

3. GOALS AND APPROACHES

Our approach uses the following constraints:

- The metric or combination of metrics must be computed automatically from the user interface metadata (i.e. data about the data).

- The metric or combination of metrics provides a value we can use automatically in test case generation or evaluation. GUI metrics should guide the testing process in aiming at those areas that require more focus in testing. Since the work reported in this paper is the start of a much larger project, we will consider only single metrics. Combinations of metrics will be left to future work. A good choice of such metrics should include metrics that can relatively be easily calculated and that can indicate a relative quality of the user interface design or implementation.

- The GUI structure can be represented through a hierarchical tree. However, there are some many-to-many relations that exist in some GUI components which may violate this assumption.

Below are some metrics that are implemented dynamically as part of a GUI test automation tool [17]. This paper is a continuation of the paper in [18] in which we introduced some of the suggested GUI structural metrics.

We developed a tool that, automatically, generates an XML tree to represent the GUI structure. The tool creates test cases from the tree and executes them on the actual application. More details on the developed tool can be found on authors' references.

- Total number of controls or Controls' Count (CC). This metric is compared to the simple Lines of Code (LOC) count in software code metrics. Typically a program that has millions of lines of code is expected to be more complex than a program that has thousands. Similarly, a GUI program that has large number of controls is expected to be more complex relative to smaller ones. Similar to the LOC case, this is not entirely true because some controls are easier to test than others.

It should be mentioned that in .NET applications, forms and similar objects have GUI components. Other project components such as classes have no GUI forms. As a result, the controls' count by itself is irrelevant as a reflection for the overall program size or complexity. To make this metric reflects the overall components of the code and not only the GUI; the metric is modified to count the number of controls in a program divided by the total number of lines of code. Table **1** shows the results.

The controls'/LOC value indicates how much the program is GUI oriented. The majority of the gathered values are located around 2%. Perhaps at some point we will be able to provide certain data sheets of the CC/LOC metric to divide GUI applications into heavily GUI oriented, medium and low. A comprehensive study is required to evaluate many applications to be able to come up with the best values for such classification.

Table 1. Controls' /LOC Percentage

Applications Under Test (AsUT)	CC	LOC	CC/LOC
Notepad	200	4233	4.72%
Calculator	58	196388	0.03%
CDiese Test	36	5271	0.68%
FormAnimation App	121	5868	2.06%
winFomThreading	6	316	1.9%
WordInDotNet	26	1732	1.5%
WeatherNotify	83	13039	0.64%
Note1	45	870	5.17%
Note2	14	584	2.4%
Note3	53	2037	2.6%
GUIControls	98	5768	1.7%
ReverseGame	63	3041	2.07%
MathMaze	27	1138	2.37%
PacSnake	45	2047	2.2%
TicTacToe	52	1954	2.66%
Bridges Game	26	1942	1.34%
Hexomania	29	1059	2.74%

The CC/LOC metric does not differentiate between an organized GUI from a distracted one (as far as they have the same controls and LOC values). The other factor that can be introduced to this metric is the number of controls in the GUI front page. A complex GUI could be one that has all controls situated flat on the screen. As a result, we can normalize the total number of controls to those in the front page. 1 means very complex, and the lower the value, the less complex the GUI is. The last factor that should be considered in this metric is the control type. Controls should not be dealt with as the same in terms of the effort required for testing. This factor can be added by using controls' classification and weight factors. We may classify controls into three levels; hard to test (type factor =3), medium (type factor =2), and low (type factor=1). Such classification can be heuristic depending on some factors like the number of parameters for the control, size, user possible actions on the control, etc.

- **The GUI tree depth.** The GUI structure can be transformed to a tree model that represents the structural relations among controls. The depth of the tree is calculated as the deepest leg or leaf node of that tree.

We implemented the tree depth metric in a dynamic test case reduction technique [19]. In the algorithm, a test scenario is arbitrary selected. 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 previously described reduction process. The selection of the number five for test scenarios is heuristic. The idea is to select the least amount of test scenarios that can best represent the whole GUI.

The tree depth is relevant to structural testing. It is directly proportional to the complexity of the GUI. GUI structure is hierarchical and as a result, the tree depth (i.e. max height, represents the maximum number of levels that exists in the application's user interface. The tool pareses all tree paths from the start to the end and counts the longest path.

- **The structure of the tree.** A tree that has most of its controls toward the bottom is expected to be less complex, from a testing viewpoint, than a tree that has the majority of its controls toward the top as it has more user choices or selections. If a GUI has several entry points and if its main interface is condensed with many controls, this makes the tree more complex. The more tree paths a GUI has the more number of test cases it requires for branch coverage.

Tree paths' count is a metric that differentiate a complex GUI from a simple one. For simplicity, to calculate the number of leaf nodes automatically (i.e. the GUI number of paths), all controls in the tree that have no children are counted.

Table 2. Tree Paths, Depth and Average Tree Edges Per Level Metrics

Application Under Test (AUT)	Tree Paths' Number	Edges/Tree Depth	Tree Max Depth
Notepad	176	39	5
CDiese Test	32	17	2
FormAnimation App	116	40	3
winFomThreading	5	2	2
WordInDotNet	23	12	2
WeatherNotify	82	41	2
Note1	39	8	5
Note2	13	6	3
Note3	43	13	4
GUIControls	87	32	3
ReverseGame	57	31	3
MathMaze	22	13	3
PacSnake	40	22	3
TicTacToe	49	25	3
Bridges Game	16	6	4
Hexomania	22	5	5

- Choice or edges/ tree depth. To find out the average number of edges in each tree level, the total number of choices in the tree is divided by the tree depth. The total number of edges is calculated through the number of parent-child relations. Each control has one parent except the entry point. This makes the number of edges equal to the number of controls-1. Table **2** shows the tree paths, depth and edges/tree depth metrics for the selected AUTs.

The edges/tree depth can be seen as a normalized tree-paths metric in which the average of tree paths per level is calculated.

- **Maximum number of edges leaving any node.** The number of children a GUI parent can have determines the number of choices for that node. The tree depth metric represents the maximum vertical height of the tree, while this metric represents the maximum horizontal width of the tree. Those two metrics are major factors in the GUI complexity as they decide the amount of decisions or choices it can have.

In most cases, metrics values are consistent with each other; a GUI that is complex in terms of one metric is complex in most of the others.

To relate the metrics with GUI testability, we studied applying one test generation algorithm developed as part of this research on all AUTs using the same number of test cases. Table **3** shows the results from applying AI3 algorithm [17-19] on the selected AUTs. This algorithm guarantees a unique test scenario in every new test case through comparing the new test case with all generated test cases in the file. Some of those AUTs are relatively small, in terms of the number of GUI controls. As a result many of the 50 or 100 test cases reach to the 100 % effectiveness which means that they discover all tree branches. The two AUTs that have the least amount of controls (i.e WinFormThreading and Note2); achieve the 100% test effectiveness in all three columns.

Comparing the earlier tables with Table **3**, we find out that for example for the AUT that is most complex in terms of number of controls, tree depth and tree path numbers (i.e. Notepad), has the least test effectiveness values in the three columns; 25, 50, and 100.

Selecting the lowest five of the 16 AUTs in terms of test effectiveness (i.e. Notepad, FormAnimation App, WeatherNotify, GUIControls, and ReverseGame); two are of the highest five in terms of number of controls, tree paths' number, and edges/tree depth; three in terms of controls/LOC, and tree depth. Results do not match exactly in this case. (i.e. not all expected results came true when we compare this metric with the earlier ones). However, it gives us a good indication of some of the GUI complex applications. Future research should include a comprehensive list of open source projects that may give a better confidence in the produced results.

Next, the same AUTs are executed using 25, 50, and 100 test cases. The logging verification procedure [18] is calculated for each case. In this procedure, executed test cases are logged and compared with the generated test cases (i.e. the input). As a result, If the input and output files are not equal, this can be from the automatic execution process itself, or they can be actual errors (which are of the most interest, from a testing perspective). Some of those values are averages as the executed to generated percentage is not always the same (due to some differences in the number of controls executed every time). Table **4** shows the verification effectiveness for all tests.

Table 3. Test Case Generation Algorithms' Results

Effectiveness (Using AI3)			
AUT	**25 Test Cases**	**50 Test Cases**	**100 Test Cases**
Notepad	0.11	0.235	0.385
CDiese Test	0.472	0.888	1
FormAnimation App	0.149	0.248	0.463
winFomThreading	1	1	1
WordInDotNet	0.615	1	1
WeatherNotify	0.157	0.313	0.615
Note1	0.378	0.8	1
Note2	1	1	1
Note3	0.358	0.736	1
GUIControls	0.228	0.416	0.713
ReverseGame	0.254	0.508	0.921
MathMaze	0.630	1	1
PacSnake	0.378	0.733	1
TicTacToe	0.308	0.578	1
Bridges Game	0.731	1	1
Hexomania	0.655	1	1

The logging verification process implemented here is still in early development stages. We were hoping that this test can reflect the GUI structural complexity with proportional values. Of the five most complex AUTs, in terms of this verification process (i.e. Note2, CDiese, ReverseGame, Note1, and PacSnake), only one is listed in the measured GUI metrics. That can be either a reflection of the immaturity of the verification technique implemented, or due to some other complexity factors related to the code, environment, or any other aspects. The effectiveness of this track is that all the previous measurements (the GUI metrics, test generation effectiveness, and execution effectiveness) are calculated automatically without the user involvement. Testing and tuning those techniques and testing them extensively will make them powerful and useful tools.

We tried to find out whether layout complexity metrics suggested above are directly related to GUI testability. In other words, if an AUT GUI has high value metrics, does

Table 4. Execution Effectiveness for the Selected AUTs

Test Execution Effectiveness Using AI3 [17,18]				
AUT	**25 Test Cases**	**50 Test Cases**	**100 Test Cases**	**Average**
Notepad	0.85	0.91	0.808	0.856
CDiese Test	0.256	0.31	0.283	0.283
FormAnimation App	1	1	0.93	0.976
winFomThreading	0.67	0.67	0.67	0.67
WordInDotNet	0.83	0.67	0.88	0.79
WeatherNotify	NA	NA	NA	NA
Note1	0.511	0.265	0.364	0.38
Note2	0.316	0.29	0.17	0.259
Note3	1	0.855	0.835	0.9
GUIControls	0.79	1	1	0.93
ReverseGame	0.371	0.217	0.317	0.302
MathMaze	NA	NA	NA	NA
PacSnake	0.375	0.333	0.46	0.39
TicTacToe	0.39	0.75	0.57	0.57
Bridges Game	1	1	1	1
Hexomania	0.619	0.386	0.346	0.450

that correctly indicate that it is more likely that this GUI is less testable?!

4. THE GUI AUTO TOOL

A test automation tool is developed that includes all test automation processes: test case generation, execution and verification. The tool first generates an XML file from the tested application executable or binary files. The XML file represents the application model in a hierarchical tree format. The main file of the application which includes the entry method is the top entity of the tree. Each control which is called directly from the main file is a child for it. All forms that are called from the main forms are also considered children for the parent form. This is recursively repeated for every control or form children. Along with every control, the properties of the control and their values are parsed to the XML file. Several algorithms are developed for test case generation. The main goal is to generate unique test cases that may guarantee good coverage of the different paths in the tested applications.

5. CONCLUSION AND FUTURE WORK

Software metrics supports project management activities such as planning for testing and maintenance. Using the process of gathering automatically GUI metrics is a powerful technique that brings the advantages of metrics without the need for separate time or resources. In this research, we introduced and intuitively evaluated some user interface metrics. We tried to correlate those metrics with results from test case generation and execution. Future work will include extensively evaluating those metrics using several open source projects. In some scenarios, we will use manual evaluation for those user interfaces to see if those dynamically gathered metrics indicate the same level of complexity as the manual evaluations. An ultimate goal for those dynamic metrics is to be implemented as built in tools in a user interface test automation process. The metrics will be used as a management tool to direct certain processes in the automated process. If metrics find out that this application or part of the application under test is complex, resources will be automatically adjusted to include more time for test case generation, execution and evaluation.

REFERENCES

[1] I. Melody, and M. Hearst, "The state of the art in automating usability evaluation of user Interfaces", *ACM Comput. Surv. (CSUR)*, vol. 33, no. 4, pp. 470-516, 2001.

[2] LabView 8.2 Help. "User interface metrics", National Instruments. <http://zone.ni.com/reference/-enXX/help/371361B01/lvhowto/userinterface_statistics> 2006.

[3] P. Robert, "Human-computer interactions design and practice", Course Website. <http://www.csl.mtu.edu-/cs4760-/www> 2007.

[4] G. Philip, "Too many clicks! Unit-based interfaces considered harmful", Gamastura. <http://gamasutra.com/features/20060823/goetz_01.shtml> 2006.

[5] T. Comber, and J. Maltby, "Investigating Layout Complexity", *3rd International Eurographics Workshop on Design, Specification and Verification of Interactive Systems.* Belgium, 1996, pp. 209-227.

[6] T. Tullis, "The formatting of alphanumeric displays: A review and analysis", *Human Factors*, pp. 657-683, 1983.

[7] S. Balbo, "Automatic evaluation of user interface usability: dream or reality", In *Proceeding of the Queensland Computer-Human Interaction Symposium. QCHI'95*, 1995.

[8] K. Robins, "User interfaces and usability lectures", http://vip.cs.utsa.edu/-classes/cs623s2006, Course Website, 2006.

[9] F. Ritter, D. Rooy, and A. Robert, "A user modeling design tool for comparing interfaces", In *Proceeding of the 4th International Conference on Computer-Aided Design of User Interfaces CADUI'2002*. 2002, pp. 111-118.

[10] M. Deshpande, and K. George, "Selective Markov models for predicting web-page accesses", *ACM Trans. Internet Technol. (TOIT)*, vol. 4, no. 2, pp. 163-184, 2004.

[11] Ch. Farenc, Ph. Palanque, and R. Bastide, "Embedding ergonomic rules as generic requirements in the development process of interactive software", In *Proceeding of the 7th IFIP Conference on Human-Computer Interaction INTERACT'99*, 1999.

[12] Ch. Farenc, and Ph. Palanque, "A generic framework based on ergonomic rules for computer-aided design of user interface", In *Proceeding of the 3rd International Conference on Computer-Aided Design of User Interfaces, CADUI'99*, 1999.

[13] A. Beirekdar, J. Vanderdonckt, and M. Noirhomme-Fraiture, "KWARESMI1; Knowledge-based web automated evaluation with reconfigurable guidelines optimization", 2002.

[14] T. Tullis, "*Screen Design. Handbook of Human-Computer Interaction"*, Elsevier Science Publishers: The Netherlands, pp. 377-411. 1988.

[15] T. Tullis, "A system for evaluating screen formats", In *Proceeding for Advances in Human-Computer Interaction*, 1988, pp. 214-286.

[16] C. Thomas, and N. Bevan, "Usability Context Analysis: A practical guide", Version 4. National Physical Laboratory, Teddington, UK. 1996.

[17] I. Alsmadi, and K. Magel, "GUI path oriented test generation algorithms", In *Proceeding of Human-Computer Interaction Conference*. *IASTED HCI*, 2007.

[18] I. Alsmadi, and K. Magel, "GUI path oriented test case generation", In *Proceeding of the International Conference on Software Engineering Theory and Practice* (*SETP07*), 2007.

[19] K. Magel, and I. Alsmadi, *GUI Structural Metrics and Testability Testing*, IASTED SEA 2007.

Received: June 11, 2008 Revised: May 19, 2009 Accepted: May 19, 2009