Universal Journal of Computer Science and Engineering Technology
1 (1), 41-49, Oct. 2010.

Automatic Model Based Methods to Improve Test Effectiveness

Izzat Alsmadi, Samer Samarah, Ahmad Saifan and Mohammed G. AL Zamil
Department of Computer Information Systems
Yarmouk University
Irbid, Jordan
ialsmadi@yu.edu.jo, samers@yu.edu.jo, Ahmads@yu.edu.jo, Mohammedz@yu.edu.jo

***Abstract*— Software testing covers a large percent of the software development expenses. However, formal methods are applied, usually, to improve or ensure the correctness of the requirements, design, code, or testing. In order to utilize formal methods particularized to different cases, the subject matter needs to be written in a formal language or syntax. In this research, several model based methods are investigated and experimented in order to reduce testing expenses, improve test coverage, and the effectiveness of the testing process.**

Formal models are generated from the application during runtime. For this purpose a tool is developed to automatically derive the formal syntax from the application at runtime. Later on, the formal model is used in improving test effectiveness. In addition, the model is used to find some possible dynamic problems in the application that might be hard to be discovered by traditional testing methods. Finally, a test monkey tool is proposed in order to test the application for deadlock or progress problems and test the application ability to reject invalid test cases as well.

Keywords- software engineering, software testing, model based verification, user interface verification, Interface model, GUI specification, software verification, formal methods.

I. INTRODUCTION

Model verification implies verifying that the designed model matches or complies with its requirements or specifications. Similarly, user interface verification refers to the process of verifying that all GUI-widgets are in their expected state. For instance, if you copy a certain text in a text editor application, GUI-widgets' states change should be reflected on the paste control (to become enabled), and on the clipboard to save the copied text. In this research, we are not intended to discuss the user interface validation that is usually accomplished through users. The proposed implementation is used as a part of a full GUI test automation framework developed by the first author [16 and 17].

The main contribution of this research includes enhancing the exiting tool with the following new features: 1) the automatic generation and 2) an LTSA file from the dynamic GUI model. This file is similar to Event Flow Graph (EFG) or a GUI state diagram, which includes all possible GUI states and transitions among these states. Our work, also, concentrates on evaluating all possible types of GUI testing coverage such as: nodes, edges and path. However, several relevant metrics were also proposed for the same purpose.

Formal verification is accomplished through making assertions about the design, by formulating properties based on the specification of the system in addition to applying mathematical and logical rules to prove that the design meets these properties. Verification can support testing and save the effort of exhaustively test the implementation. As it is always stated, "testing can only show the presence, but not the absence of errors".

Usually, interface designers use informal or ad-hoc techniques such as mock-ups or prototypes to define or describe the user interface. In some cases, such techniques are incomplete and/or vague; leading developers and users to interpret them in different ways.

Adequate GUI testing is resource consuming and infeasible; it is very difficult and expensive to automate this process. However, bit-by-bit verification has long been abandoned. Modern development languages allow us to extract and interrogate GUI information from their executables. Therefore, verification of some properties can support the testing process and provide another channel to improve the overall testing coverage. This way, we can overcome the complexity resulted from automating GUI testing. Such complicated process faces challenges in generating, executing, and verifying test cases. Neither process can claim completely proving of the correctness of the user interface.

The generic steps that are involved in the application of a model checker are [40]:

1) *Obtain a model of the source code.*
2) *Obtain the requirements to be checked.*
3) *Perform model checking step (automated).*
4) *Evaluate any error traces that the model checker generates, determine if the error is in the code or the requirements, and repeat the process if needed.*

Corresponding Author: Izzat Alsmadi, Department of Computer Information Systems, Yarmouk University, Irbid, Jordan

II. RELATED WORK

In the literature of software testing, many research articles discussed the general usage of formal methods in addition to the utilization of model checkers in this field [1,2,3,4,5,6,7,8,9,10,11,12,13,14, and 15]. In [1, 4, and 15], authors discussed the application of model checkers to improve coverage in the automatic generation of test cases. One of the main complains about the formal methods is that it delay the software development process and consume the already limited resources. Using formal methods for the automatic generation of test cases may compensate for such resource consumption.

Model checking has been widely applied to verify different types of real-time systems including: distributed systems [2], Motorola cell phones user interfaces [3], and embedded systems [5] by detecting some incidents of errors that can be rarely found by other techniques.

In general, to use a formal model checker, the user must begin by defining a formal model of the application in which the sources for these models are the requirements, design, or the code itself. In this paper, the formal model is driven from the application itself, which is the opposite of the usual use of formal methods for the purpose of generating the code from the formal model.

The automatic generation of the design or test cases using formal methods is another major relevant research area that has been investigated by many articles such as those in [2, 6, 7, 8, 9, 10, 11, 13, 14, and 15]. The major contribution is to find all system's possible states and transitions among them to model a system. However, some systems such as user interface, interactive, or distributed systems can generate a large number of possible states (i.e. state-explosion problem), which might affect the simplicity of developing the system and make the modeling a tedious task.

In order to verify GUI requirements, they have to be formally described (formal rather than detailed specifications). Such specifications can be extracted from GUI guidelines such as: workflows, windows, common actions, buttons, pop-up menus, drag and drop features, item selections, layouts, and dialog guidelines. Many research papers have proposed and discussed methods to automate GUI generation through formal specifications or GUI languages [22, 23, 25, 29, 31, 34, 35, 37, 38, and 39]. A major drawback of these methods is that they require a relatively long learning curve that does not fit the user interface in unstable and continuously evolved environment. Thus, companies tend to prefer paying more for testing than investing in these resources for GUI formal verification.

In [37, 38], authors investigated generating test oracles using formally-specified GUI events. The idea is to identify the next state (state 2) depending on previous state as well as the current event (state1-event1- state2). Consequently, automating the creation of event's flow-graph and expecting the test-case results become possible. This assumption might involve some abstraction, since it assumes that the same event on the same state causes the transition to a similar state is always true. For example, assume that clicking the event-button "save" in an open document causes the transition to another state (e.g. a document is saved in a certain location with a certain name). The name or the destination can be different. However, we assume that the final state is the same. Therefore, we can use the initial state information to verify the next state results.

In addition to the above methods and techniques, researchers investigated different methods to extract test cases from design models such as UML [28]. Similarly, in [18] described the Integrated Design and Automated Test-case Generation (IDATG) environment for GUI testing. IDATG supports the generation of test cases based on a user-task model as well as a user-interface behavioral model. Another important model for designing user interface is GOMS [24, 27]. GOMS has been applied [30, 32] to utilize operators, methods and selection rules for the purpose of designing and evaluating user-interfaces. Specifically, GOMS analyzes the user complexity of interactive systems and models the user behavior. Prediction of such complexity is a major drawback of GOMS since it is only valid for expert users. Moreover, GOMS does not take into account the behavior of non-expert users; users who just learning the systems, or intermediate users who make occasional errors; GUI aims to achieve maximum usability for different types of users.

III. GOALS AND APPROACHES

In order to verify a software model or design against its specifications, the specifications have to be formulated formally. Since specification is an integral part of the verification process as it represents the behavioral properties of systems, we cannot ignore it. However, the core question in this phase is: How can we formulate a verifiable set of GUI specifications in an appropriate format?

A. GUI graph and Test coverage

An application GUI can be formally defined by:

$$\boldsymbol{G = (C, E, A, V, N, X)} \quad (1)$$

where ***C*** represents all GUI control components (whether they are containers such as forms, panels, frames, group-boxes, or individual components such as: buttons, textboxes, labels, etc.). A control can be invoked by another control in the upper or adjacent level. ***E*** represents the Edges between components where there are certain – definite – number of edges. Each edge connects two consecutive components. The relation between the number of controls, the GUI paths, and the nodes in the GUI graph is defined as: every two nodes or controls of ***G*** are joined by a unique path. Therefore, the number of controls is equal to the number of edges +1. This implies that the edge coverage can be achieved if the number of test cases is equal to the number of GUI controls -1. In other words, edges, nodes and control coverage requires nearly the same number of test cases. In a GUI graph, we do not have statement coverage; rather, control (i.e. GUI tree components, their events and interactions) coverage is the alternative.

Each node in the GUI represents a window and encapsulates all the widgets, properties or attributes, and values in that window [41]. Specifically, there is an edge from control ***C1*** to control ***C2*** if the window represented by ***C2*** is opened by

performing an event in the window represented by ***C1***. In other words, ***C2*** will not be visible while ***C1*** is invisible, or if ***C2*** is triggered at time ***T2*** then ***C1*** would have been triggered in time ***T1*** which is earlier than ***T2***. This restriction in states or space availability is important to reduce possible states in a GUI; reducing the effect of state-explosion problem.

A GUI path is any path that starts by an entry control or node and end in a leaf node. Usually, each GUI application has one entry node. Exit or leaf nodes can be controls in any form; web page or container called through the entry form or page. As such, the number of GUI paths can vary and can not be calculated through knowing only the number of edges and controls. In software test automation, each GUI path can be tested by one more path to achieve coverage (depending on the type of coverage we are evaluating).

In the previous GUI graph formal formula, ***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. Finally, ***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. However, experimental tests will be developed to change any representative among the six parameters and evaluate the GUI states' verification algorithms to detect those changes.

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

- 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 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 observed, through testing a large number of real-time applications, that this type of changes occurs infrequently, despite the fact that the impact of such changes maybe high.

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

- 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 implies that the focus, here, is on the values of the attributes. The majority of dynamic state changes occurred as a result of such types of actions. However, one might argue that a value change of an attribute should not cause a state change. For instance, 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 situation as a state change, specially where a test automation tool is used to test a user interface. In such scenario, the test automation tool needs to know that the location of the control is modified, needs to accommodate for this change, and expects it in the new location. However, some other control attributes' modifications such as the modification of the text of a textbox control are 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 among controls that is intended to show the reachability among them. A GUI path can be defined as several edges that start from an entry point and ends in an exit or leaf point. Fig. 1 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.

- 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 points are 1) it is the entry to access all controls and all edges and 2) it is considered as the parent of all parents in the application. This is why all GUI paths in Fig. 1 starts with "FrmDataDisplay" which is the entry form.

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

Figure 1. GUI paths sample for an AUT

- Exit points [***X***]. Unlike entries, there are several exist points. Fig. 1 shows many leaf controls that can be considered as exit points for this application. The algorithm, which 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 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 on 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.

We developed a tool for the automatic generation and execution of GUI test cases [16] [17]. The tool generates a GUI tree model from the implementation using its metadata. Test cases are generated using several algorithms that consider the tree paths' coverage. The goal is to test each path in the generated tree (i.e. branch coverage). Execution is accomplished through running some API's that simulate user actions. The execution process uses the output of the dynamically generated test cases as input and logs the all events and compares them with the original test scenarios.

A GUI graph is a directed graph in which two types of controls or nodes have special properties. The entry node has zero in-nodes (nodes that are pointing to it), and many out-nodes (nodes referred to by it). In general all other nodes can be reached from the entry node. The exit node has a zero out-node(s), and many nodes in the graph have a directed path to that node. The graph may have two types of nodes—process nodes or nodes that have only one out-node link, and predicate nodes that have two or more out-nodes. A path is a sequence of successive edges, from the entry node to the exit node. A metric is designed and developed to calculate process and predicate nodes in tested applications. Table 1 shows the results of this metric.

TABLE I. GUI CONTROLS METRICS

AUT	No of controls	No of leaf controls	Max No of children
1	43	26	8
2	187	145	31
3	35	27	14
4	90	67	23

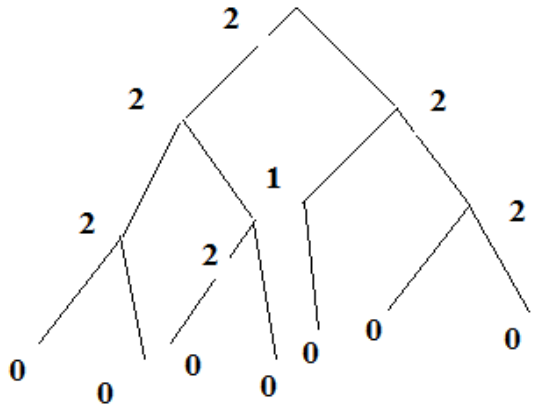

Figure 2. A tree with direct children calculation

Table 1 shows controls' structural distribution for relatively small applications. The distribution in Table 1 shows that the majority of controls are leafs; with zero out-links (more than 70 % of controls in all tested applications). On the other hand, one or few controls have a relatively large number of child nodes (i.e. out nodes). The numbers show the direct child only (i.e. including only direct child and excluding all grand children). This can be the forms in Windows-based applications or the web pages in web applications.

Knowing that most of the nodes are leaf-nodes can help us achieve path testing with relatively small number of test cases; we mentioned earlier that visiting a node once may guarantee node or edge coverage but not path coverage. The only exception to this situation is visiting the leaf-node only one time can guarantee all three types of coverage at the same time (i.e. node, edge, and path). However, thorough studies should be conducted to verify this claim. For example, if statements, branches, and path types of coverage are correlated in structural testing with nodes, edges and paths coverage in GUI testing, then we can infer that statement coverage does not guarantee branch coverage and path coverage does not guarantee decision or branch coverage.

Fig. 2 shows that the actual number of paths (from entry node to exit nodes) is 7. This can be simply computed by calculating all exit nodes. Formally, any path should contain an entry node and an exit node. More precisely, any legal path should start with an entry node and ends with a leaf node.

B. Using a model checker in GUI testing

As the name implies, Model-checkers are formal-method tools utilized to define and verify some properties of a model; represent decision procedures for temporal propositional logic.

Particularly, a model-checker is suited to show properties of a finite state-transition system [33]. However, the model contains some important requirements such as progress and safety requirements. Safety requirements are intended to check the absence of deadlocks and similar critical states that can cause system crash. Examples of well-known model checkers include: SPIN, NuSMV, PRISM, LTSA, Blast, Chess, Pathfinder, MRMC, TLC, TLV, etc. In this research, the model checker LTSA is employed to serve the goal of model check some selected system properties [43].

The proposed tool, in this paper, automates the transformation of an input file to the LTSA modeling specification during run-time of the application. The goals of generating the models at run-time are: 1) to test some properties of the system (such as safety, deadlock, progress, etc.) and 2) to improve test coverage and test effectiveness using model checking.

Using LTSA, we define some properties to be checked in order to verify the correctness and safety of the model. The verification of the implementation model, rather than the design model, is expected to expose different issues. While the design model is closer to the requirements, it is more abstract and might cause some difficulties for testing. On the other hand, the implementation model is closer to testing and is expected to be easier to test and expose more relevant errors. These errors could be a reflection of requirement, design, or implementation problems.

Intuitively, the conformance relation holds between an implementation and a specification if, for every trace in the specification, the implementation does not contain unexpected deadlocks. In other words, if the implementation refuses an event after such a trace, the specification is expected also to refuse this event [40].

In order to facilitate the verification task, the LTSA files are generated dynamically without user involvement. A formal definition of an event in LTSA is defined as the following:

$$\boldsymbol{S1} = (\boldsymbol{e1} \rightarrow \boldsymbol{S2}),$$

where ***S1*** is the initial state, ***S2*** is the next state, and ***e1*** is the event that causes the transition. We formalize the names of the states to dynamically generate the LTSA file. For example,

$$\boldsymbol{FILE} = (\boldsymbol{save} \rightarrow \boldsymbol{ok} \rightarrow \boldsymbol{saveokFILE})$$

$$\boldsymbol{FILE} = (\boldsymbol{save} \rightarrow \boldsymbol{cancel} \rightarrow \boldsymbol{savecancelFILE})$$

where the next state's name is the combination of the event(s) and the initial state. This implies that the same event on the same state is intended to transit the model into the same next state. For instance, saving different data to a file makes the same consequences of the event on the affected objects. Fig. 3 and 4 show simple LTSA demonstrations for possible states in a Word processing application. Fig. 3 shows a simple Font style options LTSA demonstration (the possible states transition from selecting the font style events: bold, italic, and underline). Similarly, Fig. 4 shows File menu options states.

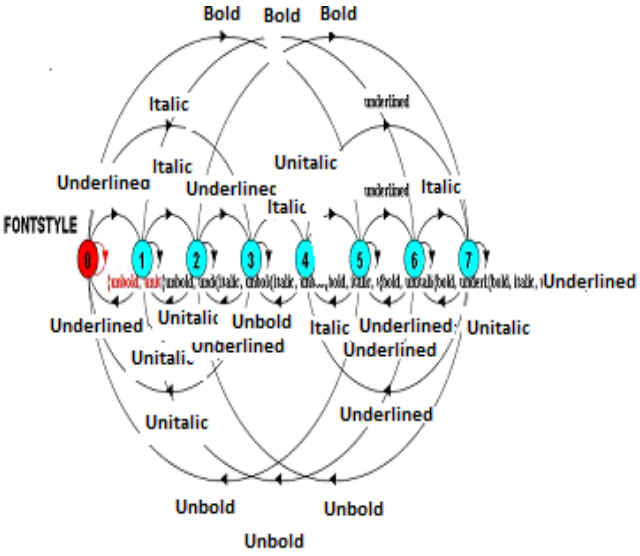

Figure 3. An LTSA font style example

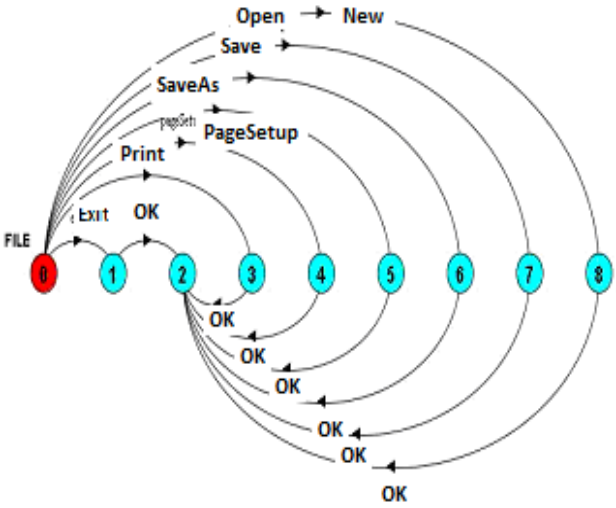

Figure 4. LTSA representation for File menu events

After generating the LTSA file (with extension .lts) for the GUI model, some selected design security properties are verified using the model checker. The tool can generate this file from the GUI tree automatically. Some examples of those security properties that can be checked include: deadlock, progress, and safety. For example, in deadlock the tool can check whether there is a GUI control that is not reachable (this can happen very often in automatically generated GUI controls). Moreover, the control properties visibility and reachability are checked and compared in the model checker in addition to the testing stage for evaluation and correlation as well.

The following properties are examples of some security specifications to be checked using LTSA. The examples show the edit-cut-copy –paste –undo properties:

```
Set Edit events={cut,copy,paste,undo}EDIT=(cut->CUTEDIT|
copy->COPYEDIT|
{paste,undo}->EDIT),CUTEDIT=(undo->EDIT|  {cut,copy}->CUTEDIT|
paste->PASTECUTEDIT),COPYEDIT=(undo->EDIT|
{cut,copy}->COPYEDIT|
paste->PASTECOPYEDIT),PASTECOPYEDIT=(undo>EDIT|
paste->PASTECOPYEDIT),PASTECUTEDIT=(undo->EDIT|
paste->PASTECUTEDIT).
Property PASTE =({cut,copy} ->paste -> PASTE)
```

We combine the edit process with a check property to make sure that the application should not allow the event "paste" to be satisfied before one of the two events: copy and cut. Such properties can be validated by the model checker on the actual application.

To carry out the experimental test on a set of application, model-verification is used first to test if there are any unreachable GUI components, which represents a common problem in GUI or web applications. Specifically, in some GUI forms or containers, there are some components that are considered dead as they will never be visited. Similarly, in web applications dead links are links that are not reachable. Consequently, model checkers define properties (such as the one defined previously for "paste") to represent constraints on certain nodes or links reachability. For example, in the "paste" property, the node "paste" has a guard constraint that it will be only reachable through "cut" or "copy" nodes. For realistic purposes, such constraint may not need to be checked for all nodes as it can be only necessary for few ones.

1. In order to utilize model verification techniques in test automation, we implemented an automatic model verification method that is expected to be always (or usually) true for the majority of applications. Here is an explanation for this method. Automatically test if there is any unreachable node. Theoretically, any node that is not a start up node and can not be reached by a start up node is an unreachable node. The "parent" information that exists in each control can be utilized to test if there is an unreachable node. This information can divide controls into the following types:

    a. An entry node: In this node, the "parent" information will be empty as it does not have a parent. As an entry node, it will have one or more children. It is impractical to have a node with no parent and no child(ren). Each control has one parent only as the property is a one-to-one property.

    b. An intermediary node. This node is neither an entry nor a leaf node. As such, it has parent and child(ren).

    c. A leaf node. This node has a parent but with no child(ren).

According to these rules, and in order to test the progress in an application, an entry node that does not have a child is a faulty node, a node that does not have a parent is a faulty node if not a leaf node and so on. A method is developed to classify nodes into these 3 types, then, test for progress and deadlocks accordingly.

Besides their usage in testing and model verification, those rules or constraints can help in GUI states reduction which is very necessary given the large amount of possible states in any GUI application.

C. Monkey Testing

Monkey testing is a random test case generation and execution process usually accomplished through automated tools. These test cases perform mouse clicks on the screen or keystrokes on the keyboard randomly. Also, they are used usually to test issues such as robustness or resources' usage, regression testing, etc.

In contrast to using formal verification in testing, another application is developed to randomly generate test cases without observing the parent-child properties and the GUI tree structure. Those test cases are then executed on the application to see if there are any deadlock situations. Such test processes are used by many software companies to test applications robustness. Despite the fact that such tools may generate invalid sequences that can never be experimented by real users, however, they are useful from testing perspectives.

The monkey testing tool that is developed automatically generates and executes the test cases. The test execution process is used for verification where a test failure is considered whenever a test execution fails. A test execution will fail if the test case includes one invalid control, invalid edge, or invalid path. As test cases are generated randomly from all application possible nodes and edges, it is possible that the automation execution process may fail to find a node or an edge (as it is not visible in the current execution scope). Besides finding deadlock or progress problems, this testing process may also find problems related to the application robustness or its ability to reject incorrect test cases or invalid test paths.

The test monkey tool is implemented in away that allows for the automatic generation, execution and verification of randomly generated test cases. We describe the tool's algorithmic steps as follows:

- The first stage is generating the GUI graph, which contains all GUI components and their location in the graph.
- Repeat the following statements while the number of loops is less than the number of test cases to generate and execute:
    - For each cycle, go to the GUI graph and select the entry node. [There is no point of making the entry selection random as it will then always cause a test case failure from the start].
    - Select randomly one GUI component from the GUI graph and try to execute it. The verification algorithm will consider the test case as success if the selected control was successfully triggered.
    - Repeat the cycle for the number of required test cases.

Alternatively, we consider selecting a complete path of controls in each test case rather than selecting only one control. In this case, the selection will not be purely random as the automatic verification process should always verify that the newly selected control is a child for the previous control. Table 2 shows the results of applying monkey testing (the first algorithm described in the pseudo code) on several selected AUTs. The coverage of the monkey testing varies based on the number of controls visible to the first form or the entry node. The number of generated test cases in each time was selected to be 25. The tested applications are open sources simple applications written in .NET and that include several Windows forms.

Several methods or algorithms are developed earlier to automatically generate test cases from the GUI graph [16 and 17]. In this paper, we modified the algorithms to work on the monkey testing where the algorithm goal is modified to generate any test cases rather than verifying that the test cases are always valid and new which was the goal of the original algorithms. Results in Table 2 and Table 3 are using one of those algorithms.

TABLE II. MONKEY TESTING COVERAGE (VALID TEST CASES)

AUT	No of test cases	No of valid test cases	Monkey coverage %
1	25	8	32
2	25	13	52
3	25	16	64
4	25	6	24

TABLE III. GUI NODES OR EDGES COVERAGE

AUT	**No. of nodes**	**No of test cases**	**N/E coverage %**
1	43	25	60
1	43	33	77
1	43	50	100
2	187	25	23
2	187	50	38.5
2	187	100	66
3	35	25	23
3	35	50	74
3	35	150	100
4	90	25	36
4	90	50	90
4	90	130	100

D. Test cases' results verification

The above formal expression of states is used for the verification of test cases' execution. The process verifies the results from executing those test cases and compares them to the expected ones as defined. Model checkers in this case are used to verify the test cases rather than the code. Since GUI test cases represent a sequence of controls, it seems like GUI graph paths that are generated in the design of the model checker.

A typical test case dynamically generated in the tool looks like; No, NOTEPADMAIN, EDIT, FIND, TABCONTROL1, TABGOTO, where the number represents the test case number and the list represents the consecutive sequence of controls. To formally and automatically verify the results of this test case according to the previously described rules, an algorithm is developed to formally define the next state based on the current state and the transition: For example, if we have the following state transition:

State1 =(event -> State2), then State 2 can be renamed as State1_event. If another event (event 2) occurs on State2, it can be formally defines as:

State2= (event 2 -> State3)

will be named as the following:

State1_event=(event2 -> State1_event_event2)

Here is a small demonstration of the process to define states based on events and previous states:

Notepadmain=(edit-> Edit_Notepadmain),

Edit_Notepadmain=(find -> Find_ Edit_Notepadmain),

Find_Edit_Notepadmain=(tabcontrol1-> Tabcontrol1_Find_Edit_Notepadmain),

Tabcontrol1_Find_Edit_Notepadmain=(tabgoto>Tabgoto_ Tabcontrol1_Find_Edit_Notepadmain).

FILE=(save->ok->SAVEOKFILE).

FILE=(save->cancel->SAVECANCELFILE).

set Events = {new,open,save,saveAs,pageSetup,print.exit}

FILE = (new -> NEWFILE

| open -> OPENFILE

|save -> SAVEFILE

|saveAs -> SAVEASFILE

|pageSetup -> PAGESETUPFILE

|print -> PRINTFILE

|exit -> EXITFILE).

One problem with this approach is that names can get very large in later states. Another assumption, which may not be true all the time, is that the same event occurred on the same state should make transition to the same state. This may depend on the way similar transitions or states are defined. Fig. 6 shows a snapshot of an LTSA file for an application that is automatically generated. The words that are or start with capital letters (also shown in blue) represent states while small letter words represent transitions.

```
Test1=(frmdatadisplay->groupbox1->lsttables->FRMDATADISPLA
frmdatadisplay->groupbox1->FRMDATADISPLAYGROUPBOX1),
Test3=(frmdatadisplay->groupbox2->lstviews->FRMDATADISPLAY
frmdatadisplay->groupbox2->FRMDATADISPLAYGROUPBOX2),
Test5=(frmdatadisplay->groupbox3->lstfields->FRMDATADISPLA
frmdatadisplay->groupbox3->FRMDATADISPLAYGROUPBOX3),
Test7=(frmdatadisplay->frmconnect->tabcontrol1->tabaccess-
frmdatadisplay->frmconnect->tabcontrol1->tabaccess->FRMDAT
frmdatadisplay->frmconnect->tabcontrol1->FRMDATADISPLAYFRM
frmdatadisplay->frmconnect->FRMDATADISPLAYFRMCONNECT),
```

Figure 5. A snap shot from an automatically generated LTS file

E. Dealing with consecutive transitions, equivalent and intermediate states

A partial program execution for a given specific interaction with its environment can be represented by a sequence of states, which are observed at discrete intervals of time. However, the program starts from an initial or startup state, then moving from one state to the next one by executing a set of atomic program transitions. Then, it ends in a final regular, erroneous, or non-terminating state. In particular, the trace is infinite in all cases [42]. As such, equal partial program executions are these executions that start from the same initial point and end in the same final point (even if they have different transitions). Theoretically, the program can generate a large number of possible states. However, many of these states can be eliminated by different techniques. From a tester or test automation tool viewpoint, many of those test cases may not be feasible, reachable or observable. Fig. 7 shows a possible view of GUI states. This view divided the states of interest into three parts: Initial, intermediate, and final states. Initial and final states are observable and reachable. The third type is those specific intermediate states that are guaranteed to be reachable.

For instance, the following two partial program executions are equivalent:

1. MainProgram->File->Format-Copy-Paste
2. MainProgram -> Format->Copy-Paste

Such viewpoint of a system focuses only on program final states and ignores intermediate transitions or states. In this scope, it might be interesting to view test case components as elements in a set.

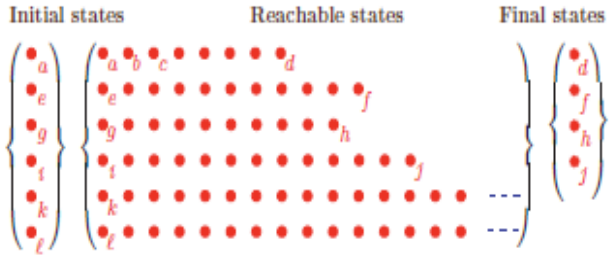

Figure 6. A possible GUI states classification [42]

1. Let S1 ={ MainProgram->File->Format-Copy-Paste}, and
2. S2={ MainProgram -> Format->Copy-Paste } be two ordered sets.

Ordered sets are sets represented as lists with the elements sorted in a standard order. If the two ordered sets start and end with the same elements, then they are considered, from equivalent states perspective, to be equal as we only focus on the first and the last elements.

F. Compare results between the model checker and the testing

In order to see the benefits of using a model checker in the overall quality of an application, some properties that can be tested among the model checkers and testing are evaluated in both stages for comparison and correlations. Examples of these properties are the visibility or the reachability of GUI controls. A model checker can test if there are deadlocks or progress problems in an application where there is a node that is not reachable (i.e. progress problem) or a node that once it is reached, the application is locked and can not exit from that node or state (i.e. deadlock). In testing, techniques are developed to evaluate such scenarios. However, it is usually easier to discover such problem earlier through the model. This means that the use of model checking can help indirectly improving node, path, or state coverage. However, it is usually difficult and time consuming to define a system formally. On the other hand, we might be able to define certain and not all system properties formally.

IV. CONCLUSION AND FUTURE WORK

In this paper, we studied GUI model and test results' verification as a complement for testing. Our experiments targeted the practice of avoiding the use of formal methods through using a lightweight formal process that can be dynamically generated without the need for using extra resources. The research demonstrates several techniques to automatically improve test coverage in the testing and the modeling stages.

In this research, several model based methods were investigated for the purpose of improving test effectiveness or coverage. Software testing is a process that occurs in later stages of the software development process, yet consumes a large amount of its resources. However, introducing such model based tool that can be formalized and tested automatically is expected to reduce the testing resources' consumption.

In future, we are going to evaluate the affectivity and the validity of the verification process. In principle, the verification process is simple and can be easily implemented which provides a promising solution for complex GUI test automation and verification processes.

GUI verification does not eliminate the need for user validation. It only provides another tool that can be added to software testing and the manual validation of user interfaces by users. Such technique can be very useful in regression testing where we need to rerun specific test cases periodically or before a new release.

V. REFERENCES

[1] G. Fraser and F. Wotawa, "Using Model checkers for Mutation-Based Test-Case Generation, Coverage Analysis and Specification Analysis", Proceedings of the International Conference on Software Engineering Advances (ICSEA'06). 2006

[2] G. Holzmann and M. Smith, "An automated verification method for distributed systems software based on model extraction". IEEE transactions on software engineering, 2002.

[3] C. Bertolini and A. Mota, "Using Probabilistic Model Checking to Evaluate GUI Testing Techniques", 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods. 2009.

[4] B. Lindstr¨om and P. Pettersson. "Generating Trace-Sets for Model-based Testing", 18th IEEE International Symposium on Software Reliability Engineering, 2007.

[5] T. Reinbacher, M. Kramer, M. Horauer and B. Schlich, "Motivating Model Checking of Embedded Systems Software", in Proc. Mechatronic and Embedded Systems and Applications (MESA08), 2008.

[6] R. Lefticaru, F. Ipate and C. Tudose, "AutomatedModel Design using Genetic Algorithms and Model Checking", 2009 Fourth Balkan Conference in Informatics. 2009.

[7] H. Margaret, J. S. Gerard and H. K. Etessami, "Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs", Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, 2001.

[8] P. Dhaussy, J. Auvray, S. de Belloy, F. Boniol, and E. Landel, "Using context descriptions and property definition patterns for software formal verification", 2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08).

[9] M. Kadono, T. Tsuchiya and T. Kikuno, "Using the NuSMV Model Checker for Test Generation from Statecharts", 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing.

[10] A. J. Ramirez and B. C. Cheng, "Verifying and Analyzing Adaptive Logic Through UML State Models", 2008 International Conference on Software Testing, Verification, and Validation.

[11] J. Staunton and J. A. Clark, "Searching for Safety Violations using Estimation of Distribution Algorithms", Third International Conference on Software Testing, Verification, and Validation Workshops, 2010.

[12] O. Pavlovi´c and H. Ehrich, "Model Checking PLC Software Written in Function Block Diagram", 2010 Third International Conference on Software Testing, Verification and Validation.

[13] L. García and S. Roach, "Model-Checker-Based Testing of LTL Specifications", 10th IEEE High Assurance Systems Engineering Symposium, 2007.

[14] M. Gligoric, T. Gvero, S. Lauterburg, D. Marinov, and S. Khurshid, "Optimizing Generation of Object Graphs in Java PathFinder", 2009 International Conference on Software Testing Verification and Validation.

[15] G. Fraser and F. Wotawa, "Ordering Coverage Goals in Model Checker Based Testing", 2008 IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08).

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

[17] I. Alsmadi and K. Magel, "An Object Oriented Framework for User Interface Test Automation". MICS07. 2007.

[18] A. Beer, S. Mohacsi, and C. Stary, "IDATG: An Open Tool for Automated Testing of Interactive Software". Proceedings of the COMPSAC '98 - 22nd International Computer Software and Applications Conference, pages 470-475 August 19-21, 1998.

[19] J. Bonnie and K. David, "Using GOMS for user interface design and evaluation: which technique?" In ACM Transactions on Computer-Human Interaction (TOCHI). Volume 3, issue 4. Pages: 287 – 319. 1996.

[20] B. A. Myers, "State of the Art in User Interface Software Tools", chapter 5.Ablex, Norwood, N.J., 1992.

[21] P. Bumbulis, "Combining Formal Techniques and Prototyping in User Interface Construction and Verification". PhD thesis, University of Waterloo, 1996.

[22] D. Carr, "Specification of Interface Interaction Objects," Proc. ACM CHI'94 Human Factors in Computing Systems Conference, pp. 372-378, Addison-Wesley/ACM Press, 1994.

[23] R. Cassino, G. Tortora, M. Tucci and G. Vitiello. "SR-task grammars: a formal specification of human computer interaction for interactive visual languages". IEEE Symposium on Human Centric Computing Languages and Environments (HCC'03) pp. 195-197. 2003.

[24] J. Elkerton, "Using GOMS models to design documentation and user interfaces: An uneasy courtship". In proceedings of INTERCHI'93. ACM, New York. 1993.

[25] J. Francis, "First steps in the retro-engineering of a GUI toolkit in the B language". ACM International Conference Proceeding Series. 2003.

[26] P. Fröhlich and J. Link, "Automated Test Case Generation from Dynamic Models". Proceedings of the 14th European Conference on Object-Oriented Programming. Pages: 472 – 492. 2000

[27] L. Hochstein, "GOMS. Course website". Available from: <http://www.cs.umd.edu/class/fall2002/cmsc838s/tichi/printer/goms.html>. 2002.

[28] V. Marlon, L. Johanne, B. Hasling, R. Subramanyan and J. Kazmeier. "Automation of GUI Testing Using a Model-driven Approach". International Conference on Software Engineering. Proceedings of the 2006 international workshop on Automation of software test. Shanghai, China, Pages: 9 – 14. 2006.

[29] M. G. Williams and V. Begg. "Translation between Software Designers and Users". Communication of the ACM. p. 102 – 103. 1993.

[30] M. C. Chuah , B. E. John and J. Pane, "Analyzing graphic and textual layouts with GOMS: results of a preliminary analysis". In proceeding of the conference companion on human factors in computing systems. USA. Pages323-325. 1994.

[31] C. Rouff, "Formal specification of user interfaces". ACM SIGCHI Bulletin. Pages: 27 – 33. 1996.

[32] E. Schlungbaum, "Model-based User Interface Software Tools, Current state of declarative models". GIT-GVU-96-30 November 1996.

[33] J. Schumann, "Automated theorem proving in software engineering". Springer. 2001.

[34] B. E. Sucrow, "Formal specification of human-computer interaction by graph grammars under consideration of information resources". Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE) Page: 28. 1997.

[35] B. E. Sucrow, "Refining Formal Specifications of Human Computer" Interaction by Graph Rewrite Rules. Springer Berlin / Heidelberg. 1998.

[36] V. Tschaen, "Model-based testing of reactive systems". Springer Berlin / Heidelberg. <www.irisa.fr/vertecs/Publis/Ps/2005-Test-Chap-Book.pdf>. 2005.

[37] Q. Xie and A. M. Memon, "Studying the Characteristics of a .Good. GUI Test Suite". Department of Computer Science, University of Maryland, College Park.

[38] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications". ACM Transactions on Software Engineering and Methodology (TOSEM) .Volume 16 , Issue 1 (February 2007) .Article No. 4. Year of Publication: 2007.

[39] Y. Ait-Ameur and M. Baron, "Formal and experimental validation approaches in HCI systems design based on a shared event B model". International Journal on Software Tools for Technology Transfer (STTT). Springer Berlin / Heidelberg. 2006.

[40] H. Smith, J. Holzmann, and K. Etessami, "Events and Constraints: A Graphical Editor for Capturing Logic Requirements of Programs". Fifth IEEE International Symposium on Requirements Engineering (RE'01). 2001.

[41] A. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces". Ph.D. thesis, Department of Computer Science, University of Pittsburgh. 2001.

[42] P. Cousot, "Abstract Interpretation Based Formal Methods and Future Challenges", LNCS 2000.

[43] LTSA - Labeled Transition System Analyzer, available from: http://www.doc.ic.ac.uk/ltsa/, 2009.