SPECIFICATION-BASED AUTOMATED GUI TESTING

Andreas S. Andreou, Anastasis Sofokleous
Department of Electrical Engineering and Information Technology, Cyprus University of Technology, Limassol, Cyprus

Charis Panayi
Department of Computer Science, University of Cyprus, Nicosia, Cyprus

Keywords: GUI testing, Specification- based testing, Automatic test-case generation.

Abstract: GUI testing is currently one of the most expensive and time consuming processes in the software life-cycle and according to some estimates this cost can reach up to 50 to 70% of the whole cost of the project. This paper proposes a framework for specification-based automated GUI testing which employs a GUI analyzer for dynamic analysis and extraction of GUI object information, a system for automatic test case generation driven by Spec#, a test-case execution algorithm that executes test-cases automatically and a verifier that compares the expected with the actual result of the test. Preliminary experimental results demonstrate the efficiency and effectiveness of the framework.

1 INTRODUCTION

This work proposes an automatic GUI testing framework based on black box testing. The framework comprises a set of specialized parts: A GUI analyzer, a way to describe software specifications using a new modelling language called Spec#, a test-case generation system, a test-case execution system and test oracles.

The main contributions of this paper are three. The first is that it proposes a complete testing framework which performs all testing tasks automatically. This includes not only dynamic GUI analysis but also automatic test-case generation, execution and verification. The second contribution is that it demonstrates a relatively novel approach which uses specifications to generate the test oracles by executing them under a specified test string, thus generating the test-case and the corresponding expected result of the test quickly and accurately. The final contribution is that it utilizes a combination of libraries found in Ranorex® Studio, which allow the interaction with the AUT, and a specially designed algorithm that automates the run-time execution of the test-cases.

The rest of this paper is organized as follows: Section 2 presents other related approaches and compares their results and findings. Section 3 describes the proposed testing framework. Section 4 evaluates the present work and provides some experimental results. Finally, Section 5 concludes the paper and suggests some steps of future work.

2 BACKROUND AND RELATED RESEARCH

There are various techniques that were used over the years for automating the GUI testing process. One of the earlier techniques that tried to automate this process was record / playback (Li and Wu, 2004). This technique has two modes. In the first mode, the tester's interactions on the AUT are recorded as mouse coordinates and then are played back in the form of test cases (second mode). This technique has several flaws, the most severe of which is that in case the GUI has the tiniest change, the test cases will break. This has been later solved by replacing the record technique with the capture technique. This technique, instead of recording mouse coordinates, it captures GUI widgets and therefore solves the aforementioned problem.

In both of the above cases the vendors of these tools claim that their products automate the GUI testing procedure. In reality, the record (or capture) / playback technique is not automatic. It requires a lot

S. Andreou A., Sofokleous A. and Panayi C..
SPECIFICATION-BASED AUTOMATED GUI TESTING.
DOI: 10.5220/0003548103180323
In *Proceedings of the 13th International Conference on Enterprise Information Systems* (ICEIS-2011), pages 318-323
ISBN: 978-989-8425-55-3

of manual efforts by the tester since the test cases are manually created and, moreover, the testers are continuously interrupted by these tools so as to insert verification checkpoints.

Paiva et al. (2005) present an extension of SpecExplorer® that adds the ability to (i) gather information about the GUI objects and (ii) generate a .NET assembly which facilitates simulation of user actions. In our approach, we have employed libraries available in the Ranorex® Studio (Ranorex, 2009) that enabled us to simulate user actions, such as mouse clicks, on the AUT. In Barnett et al. (2003) the authors propose an environment for model based testing with AsmL (Abstract State Machine Language) that supports semi – automatic parameter generation, call sequence generation and conformance testing. In their approach, AsmL is used for the generation of Finite State Machines by exploring the state space of the AUT. A technique to collect information about GUI objects can be found in (Memon et al., 2003), which utilizes reverse engineering techniques to obtain GUI object information. The authors here developed a tool called GUI Ripper, which incorporates a dynamic process that automatically traverses the GUI of the AUT by opening all of its windows and then extracting the information about each GUI object. In our approach, a similar ready-made tool has been utilized in order to obtain the necessary GUI object information. This tool is called Ranorex® Spy, and it dynamically extracts all visible GUI information. Memon et al. (2005) present a framework based on a GUI Ripper called DART, which automates the tasks needed in order to perform smoke tests. The authors make use of event flow graphs in order to represent GUI components and an integration tree to identify interactions among components. The tool presented is not fully automated as it needs testers' interaction to verify and modify the structure of the GUI as it is presented by the GUI Ripper, as well as to define a matrix for the test cases length that are to be executed. In, an approach is described where planning techniques from Artificial Intelligence are exploited in order to generate test cases for GUI systems as sequences of events. A different approach for the generation of test cases can be found in (Briand and Labiche, 2002). In this paper the authors discuss the use of UML diagrams for generating test-cases. They support that the existence of functional system test requirements, and in particular of use-case diagrams, class diagrams and collaboration diagrams, is sufficient design information to generate test cases, test oracles and test drivers. The same point of view about the use of UML diagrams to generate test cases is shared in (Kim et al., 1999). In this paper the authors present another technique for the generation of test cases out of UML diagrams. According to their study, control flow can be identified by transforming UML state diagrams into EFSMs (Extended Finite State Machines) and data flow can be identified by transforming these EFSMs into flow graphs. Finally, they use these flow graphs to generate test cases. Nevertheless, they do not support their arguments with an automated environment as a proof of concept. In (Edwards, 2001) a general strategy for automated black box testing is presented, where a specification language called RESOLVE is used with pre- and post-conditions attempting to describe sufficiently the AUTs' behavior. In (Tahat et al., 2001) a conversion of SDL (Specification and Description Language) into an EFSM model is described, where the latter becomes input to a black box test generator. In (Krichen and Tripakis, 2004) a testing framework based on timed automata for conformance testing is proposed, where the authors assume that the specifications of the AUT are given as non-blocking TAIO (Timed Automata with Input – Output) in order to avoid on the fly reachability computation, thus reducing the reaction time of the test. The work of Offutt et al. (2003) presents formal testing criteria for system level testing based on formal specifications as part of a case study to evaluate their ability to detect seeded faults. To achieve high coverage the authors employ the full predicate coverage criterion.

3 FRAMEWORK LAYOUT

The proposed framework consists of five parts (Figure 1).

The first part is essentially a combination of the Ranorex® Spy tool and a custom XML Parser. The former provides the functionality required to dynamically analyze the AUT's GUI objects, extract this information and export it to a single XML file, whereas the latter serves two purposes. Firstly, it is responsible to present this information to the user/tester in a comprehensible manner. The proposed system presents the information extracted by the XML Parser through a property grid, which is extremely helpful since, this way the user/tester can search and find the required information easily. Secondly, the XML Parser is responsible to convert the GUI object information in a certain form, so that the model described in the specifications can be easily and automatically initialized in order to have

the exact same initial state as the actual application under testing.

The second part of the proposed framework consists of the AUT's specifications. The specifications are written using a new modelling language called Spec#. Spec# is a product of Microsoft® Research and is influenced by the programming languages C# and Eiffel, and includes, among others, object invariants, pre-conditions and post-conditions. The basic characteristic that advocates in favour of the use of Spec# is the fact that the specifications described by this language are executable. In order for the components of the model and the AUT to have the same state when executing the specifications, the system initializes the modelled GUI objects to the same information that these objects have on the actual application. This is performed by automatically feeding each modelled object with the characteristics of the equivalent object of the AUT.

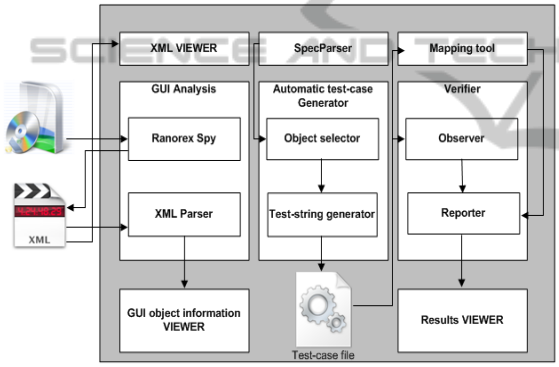

Figure 1: The architecture of the proposed framework.

The third part concerns a novel test case generation process and it is one of the novelties of the present paper. The test-cases are in fact generated through the specifications, thus enabling the test suite to serve as a test oracle as well. In order to do that, the system parses the specification's model and identifies the names of the GUI objects that are described therein. It then generates a list of objects, i.e. the GUI objects of our AUT, from which the user/tester can make a selection of the GUI objects she/he wants to perform tests on. The framework here supports the selection of the whole set of objects from the list generated, or a smaller subset. Our framework then uses a random selection algorithm by which GUI objects that are described in the specifications are randomly selected from the aforementioned list of objects to generate test-strings. The latter correspond to sequences of user driven GUI events. For efficiency purposes, we have introduced two selection criteria to the framework: 1) a user-defined number of GUI objects that is selected for testing and 2) all GUI objects that are present in the GUI are selected for testing at least once. A sequence of GUI objects (test-string) is transformed to test-cases as a set of pre-conditions, actions and post-conditions. To this end, slight modifications to the specifications model must take place beforehand. These modifications are actually print statements, in order to print the states of the modelled GUI objects to a file before and after a certain action.

The fourth part is the Mapping tool which simulates user actions on the AUT without any user intervention'. The Mapping tool parses the test-cases and identifies therein the steps of the test to be performed. As already mentioned in the previous paragraph, the test-case file is consisted by GUI events or steps, each of which comprises a triad of pre-conditions, actions and post-conditions. Next, a dedicated mapping algorithm is invoked to automate the AUT's testing procedure. Basically, this algorithm reads through the test-case file, recognizes and then executes the steps mentioned above. More specifically, when the algorithm recognizes a pre-condition or a post-condition, it acquires the state of the AUT. When it recognizes an action (or a sequence of actions), the algorithm searches for the GUI object, finds it on the AUT and then executes the specified action (or actions) directly on the actual AUT. A simple example of a GUI event (or step) can be described in a test-case as follows:

[Pre-Conditions]
button1.Pressed = False;
[Actions]
button1.Pressed = True;
[Post-Conditions]
text2.Value = "Hello World";
button1.Pressed = False;

At first (pre-condition), the algorithm will search in the AUT for the GUI object with *Control_Id* =1, which is a button, and then it will acquire the *Pressed* value of the said object (which should be equal to "False"). Then (action), the algorithm will make the *Pressed* (initial) value of the button equal to "True" (button pressed). Finally (post-conditions), the algorithm will acquire the state of the GUI, which must be equal to "True". In this example, by pressing *button1*, the message "Hello World" is displayed in the textbox with *Control_Id* = 2.

The final part of the framework is basically a validation module that works in close co-operation with the Mapping Tool. The purpose of this module

is to validate the results given by the AUT in comparison with the results given by the test oracle. Since the test-cases include the expected result, the actual execution of the test cases by the Mapping tool will provide the means for conductive comparative assessment. At this point, the module will compare the two results (expected and actual) and will present to the user/tester its findings, i.e. the information needed to enable the user/tester to verify the AUT's compliance with its specifications. The results are presented in a comprehensible manner so that the user/tester may easily identify any inconsistency between the AUT and its specification. More specifically, the tool reports each interaction our system performs on the AUT, starting from its execution up to its termination, and includes the names of all GUI objects it interacts with, the type of the interaction as well as the actual state and the state defined in the test-cases for every GUI object tested. While performing validation, our tool highlights in green every correct behaviour of the AUT and in red every incorrect behaviour, thus making the presentation of the results more user friendly.

4 EVALUATION

A proof of concept application was developed to support the experimental process (see Figure 2), which is organized as a test wizard, including the following steps:

1. Selection of the software system to be tested.
2. Selection between executing the Ranorex® Spy tool to generate the XML snapshot of the AUT and loading of a stored XML file.
3. Presentation of information related to the GUI objects participating in the AUT.
4. Definition of the system specifications.
5. Selection of the GUI objects to be tested (all or a smaller subset).
6. Tuning of the parameters of the test. The user can specify the number of test cases to be generated, the test case length and the method of generating test cases (random or manual) The manual method of test-case generation is used only for regression testing, in order for the user/tester to be able to reconstruct a test, after modifications to the source code of the AUT have taken place.
7. Execution of the test(s).

The experiments were divided into two sets. The first set of which aimed at determining, primarily whether our framework is able to perform automated GUI testing. The second set attempted to assess its effectiveness on widely-known, real-case applications. The former used the sample application (no functionality) and SpeedSim (small functionality), while the latter involved assessing the framework on the Calculator application offered in Windows. The results of the experiments of the third application showed that the proposed framework is able to interact efficiently with the AUT and achieves automation of the GUI testing process and can detect all the GUI objects that constituted the application. The SpeedSim application was used to specify the GUI objects to be tested and hence to actually model, maintaining the ability to compare the actual with the expected response. In this set of experiments, specifications were modified to be different from the actual implementation; the results of the test revealed that the behaviour of GUI objects model did not completely match that of the GUI objects in the actual AUT, thus suggesting a discrepancy between the prescribed (desired) and the actual functionality.

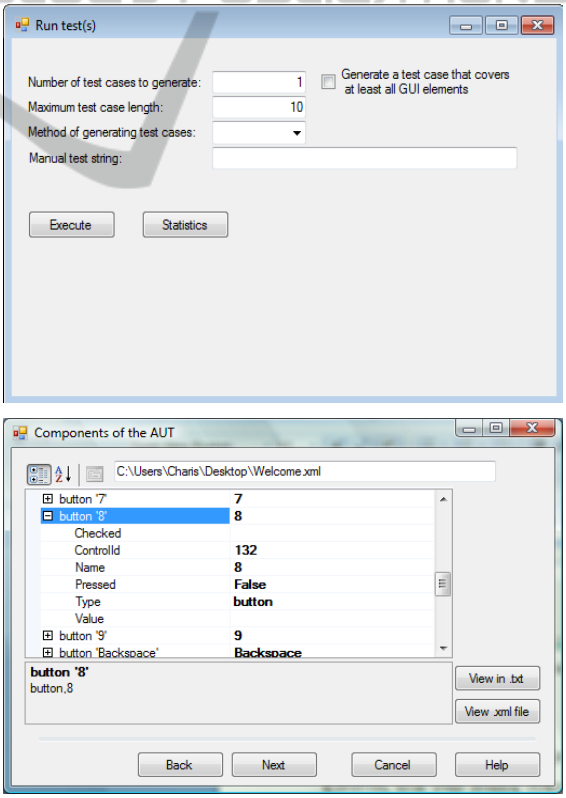

Figure 2: Screenshots of the supporting software application.

As shown in Figure 3, the framework was able to receive feedback from the AUT and therefore detect the inconsistencies between the actual and the

expected result after the simulation of GUI events; the tester is able to examine the sequence of events that took place and be informed about the errors that were identified during the actual execution of GUI events. For example, as shown in Figure 3, one may observe that when button "←" was pressed, the AUT's *textbox1235* returned a false value, i.e. it should have returned a "0" instead of a "1", something which was captured by the framework and was indicated with a red highlighted error result describing the GUI object involved, the actual and the expected values.

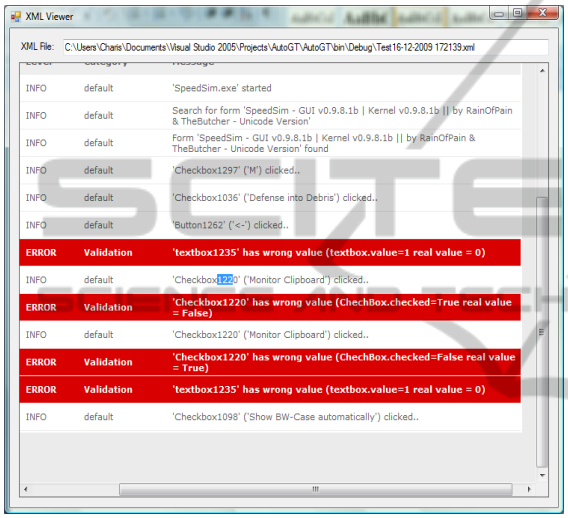

Figure 3: Results of the Speedsim test.

The first set of experiments it shows that the proposed framework in principle works successfully, has been achieved. the second part of the experiments was conducted by testing the well-known Microsoft® Calculator application and more specifically through modelling of the specifications for the buttons representing "0"-"9", "+", "-", "*", "/" and "=". The purpose of this experiment was to observe the efficiency of the framework in relation to the functionality complexity and the specifications complexity. Approximately 200 different tests were performed on the system using the Calculator application and in each test the system generated various test-strings of different lengths and for a different selection of GUI objects. To facilitate this, we used various configurations of our system parameters, that is, different test cases, different size of test-cases and different GUI object selection methods. These tests suggested that our framework is highly capable of detecting both correct and faulty execution. Observations made on the results showed that correct behaviour was observed while the equations were in the form $(x^*yx^*)^*$ and incorrect behaviour was observed while the equations were in the form $(x^*yy^*)^*$ ("*" denoting repetition) where:

$$\forall x \in N \,,\, N = \{\text{'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}\}, \quad (1)$$

$$\forall y \in O \,,\, O = \{\text{'+', '-', '*', '/', '='}\}, \quad (2)$$

Through this evaluation, inconsistencies are detected between specifications and the actual implementation (Figures 4 and 5) as the specifications are written for the calculator application through empirical use and not via any other type of formal knowledge. Overall, the experiments conducted showed that the proposed framework was capable of performing automatic testing of the Calculator application and that the actual implementation was indeed different from the one modelled in some occurrences. Figure 5 shows the results after simulating a sequence of user events shown in Figure 4.

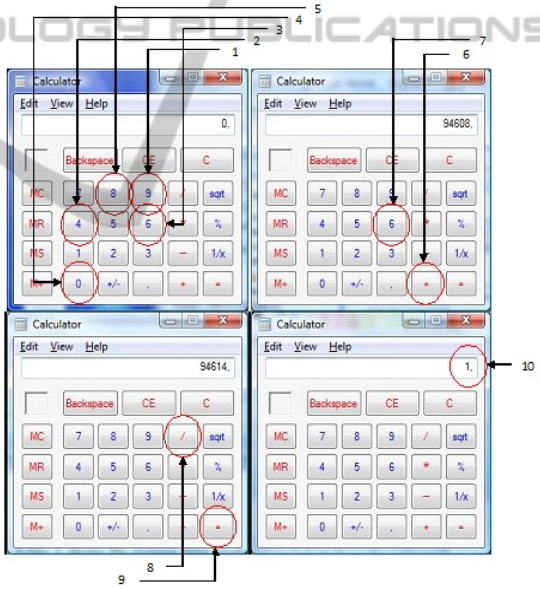

Figure 4: Execution of a test on Microsoft Calculator application.

The sequence of events specified above produced the final result "1" as displayed by the arrow 10 in Figure 4. In the generated report (Figure 5) the specifications and the implementation of the AUT are dissimilar, therefore an ERROR message in red is presented during the automatic validation process. A close inspection of the report reveals the case in which the reported error occurs, that is, when pressing any operator button (i.e. buttons representing '+'. '-', '*', '/', '=') more than once, without the intermediate pressing of numeric buttons

(i.e. buttons representing '0' – '9'), the AUT reacts differently compared to what it should have reacted according to its specifications. Therefore, the Verification module of our framework detected the difference between the modelled and the actual implementation and reported it correctly.

Figure 5: Results of the Microsoft Calculator test.

Table 1 shows the average time needed to perform the tests on the Microsoft® Calculator application. These results clearly indicate the vast increase of time when increasing the length of the test case. The time limitations are posed by the libraries of the Ranorex® Studio system, therefore these limitations currently cannot be altered unless new means that allow the interaction with the GUI without user interference are brought to light.

Table 1: Experimental measurements of time.

Test case length (GUI events)	Average Time (seconds)
5	12.069
10	23.758
15	35.438
20	46.112
50	116.854

5 CONCLUSIONS AND FUTURE WORK

This paper presented a specification-based, automatic GUI testing framework. A proof of concept application has been developed. Two different sets of tests were executed on the proposed framework. The first set verified that the proposed system can automate the GUI testing procedures, whereas the second demonstrated that efficient automated GUI testing may be achieved through the use of specifications, by successfully detecting and reporting erroneous GUI behaviour during actual execution. Future research will attempt to automate the whole process of constructing the specifications since this is the only manual work that the tester has to perform.

REFERENCES

Barnett, M., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veans, M., 2003. Model-Based Testing with AsmL.NET. *In Proceedings of the 1st European Conference on Model-Driven Software Engineering.*

Briand, L., Labiche, Y., 2002. A UML-based approach to system testing. *Journal of Software and Systems Modeling, vol. 1, no. 1, pp. 10-42.* Springer.

Edwards, S. H., 2001. A framework for practical, automated black-box testing of component-based software. *Journal of Software Testing, Verification and Reliability, vol. 11, no. 2, pp. 97-111.*

Kim, Y. G., Hong, H. S., Bae, D. H., Cha, S. D., 1999. Test cases generation from UML state diagrams *IEE Proceedings – Software, vol. 146, no. 4, pp. 187-192.*

Krichen, M., Tripakis, S., 2004. Black-box conformance testing for real-time systems. *Journal of Model Checking Software, pp. 109-126.* Springer.

Li, K., Wu, M,. 2004. *Effective GUI Test Automation: Developing an Automated GUI Testing Tool (Chapter 2).* Sybex Inc.

Memon, A., Banerjee, I., Nagarajan, A., 2003. GUI ripping: Reverse Engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*.

Memon, A., Nagarajan, A., Xie, Q., 2005. Automating regression testing for evolving GUI software. *Journal of Software Maintenance and Evolution: Research and Practice vol.17, no. 1, pp.27-64.*

Memon, A., Pollack, M., Soffa, M., 2001. Hierarchical GUI test case generation using automated planning. *Journal of IEEE Transactions on Software Engineering, vol. 27, no. 2, pp. 144-155.*

Offutt, J., Liu, S., Abdurazik, A., Ammann, P., 2003. Generating test data from state-based specifications. *Journal of Software Testing, Verification and Reliability, vol. 13, no. 1, pp. 25-53.* John Wiley & Sons.

Paiva, A. C. R., Faria, J. C. P., Tillmann, N., Vidal, R. F. A. M., 2005. A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing. In *Proceedings of ICFEM'05, Manchester, UK.*

Ranorex GmbH, 2009. *Ranorex – GUI Automation & Automated Testing Tool.* www.ranorex.com

Tahat, L. H., Bader, A., Vaysburg, B., Korel, B., 2001. "Requirement-based automated black-box test generation. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development, pp. 489-495.*