Automated Java GUI Modeling for Model-Based Testing Purposes

Pekka Aho[1], Nadja Menz[2], Tomi Räty[1] and Ina Schieferdecker[2]
[1]*VTT Technical Research Centre of Finland*
[2]*Fraunhofer FOKUS*

Abstract

Advanced methods and tools for GUI software development allow a rapid and iterative process of prototyping and usability testing. Unfortunately, even with the support of test automation tools, testing of GUI software requires a lot of manual work, especially when the application under test is changing rapidly. In this paper we present an improved method and tool support for automated test modeling of Java GUI applications for model-based testing (MBT) purposes. The implemented GUI Driver tool generates structural models combined with a GUI state model presenting the behavior of the GUI application that is executed and observed automatically. The GUI Driver tool is combined with an open source MBT tool to form a tool chain to support automated testing of Java GUI applications. The models generated by the GUI Driver are used to generate test sequences with MBT tool, and the test sequences are then executed with the GUI Driver to generate a test report.

Keywords: automatic model generation, GUI state model, GUI testing, model-based testing, tool chain

1. Introduction

Graphical User Interface (GUI) software is often large and complex, and the correctness of GUI's code is essential to the correct execution of the overall software [1]. Although advanced development methods and tools have helped to reduce the time to build software products, the time and effort required to test the products has not been significantly reduced [2].

Virtually all GUI applications today are built using toolkits and interface builders enabling a rapid and iterative process of prototyping and usability testing, crucial for achieving high-quality user interfaces [3]. Due to the iterative development process, the requirements, design and implementation of GUI software change often, increasing the time and resources required for testing.

One approach trying to improve the efficiency of software testing is model-based testing (MBT). The general idea of MBT is to create an abstract test model describing the expected behavior of the system under test (SUT), and to use some kind of tool to generate tests from this model [4]. The goal in MBT is to increase the efficiency and quality of software testing by automating the testing process, which allows decreasing the testing effort and avoiding manual error prone activities [5].

Ideally, a test model could be obtained from a design model through sophisticated model transformations. However, such tools are not available today and the approach would also rely on the availability of the design model. Currently, the models for automated testing need to be manually crafted from the specifications of the SUT, using some visual modeling tool [6]. Due to the iterative process of GUI development, manually designing and maintaining the models for MBT purposes requires too much effort, negating the increased efficiency gained from using test automation. Instead, the models should be generated automatically from the source code or by executing and observing the GUI application. The currently available solutions for automated GUI testing are not optimal and there is an obvious need for more efficient approaches.

The starting point for our work was the GUITAR tool [7], which was the most advanced open source or otherwise free tool available for automatically creating models for MBT, based on the execution and reverse engineering of an existing Java GUI application. It supports creating structural models, and generating and executing test cases for Java GUI applications. However, while using the GUITAR tool, we encountered major problems in generating models of complicated Java GUI applications. Therefore we took a slightly different approach for automated GUI test modeling and implemented GUI Driver tool to support our approach.

This paper presents our approach for automatically modeling a Java GUI application and its behavior for MBT purposes, the GUI Driver tool that we implemented to support our approach, and a tool chain we formed by combining GUI Driver with an open source MBT tool to support automated testing of Java GUI applications. Our goal is to reduce the work required to create and maintain test models, and increase the efficiency of automated testing of Java GUI applications.

The rest of the paper is organized as follows: Chapter 2 introduces background information and related work. Chapter 3 describes our approach for automated GUI modeling, the GUI Driver tool and the tool chain for automated Java GUI testing. Chapter 4 draws conclusions and addresses future work.

2. Background and Related Work

This chapter presents a short overview on the topics of GUI testing in chapter 2.1 and model-based testing in chapter 2.2, and then highlights the related work in the fields of model-based GUI testing in chapter 2.3.

2.1. GUI Testing

GUI testing is vital for quality assurance because the GUI tests are performed from the view of the end users of the application. Many times all the functionality of the application can be invoked through the GUI and therefore GUI tests can cover the entire application [8].

Because manual testing of GUI software is tedious and laborious, there is a great need for reducing the high costs by means of automated GUI testing. The most popular tools used to test GUIs are capture/replay tools [8]. Some tools record mouse coordinates of the user actions as test cases. The problem with such tools is that even a minor modification in the GUI breaks the corresponding test cases. One approach to overcome this problem is to capture GUI widgets rather than mouse coordinates. Although replay is automated, significant effort is required to create the test cases and to detect the failures. Another popular approach is to invoke the methods of underlying code as if initiated by the GUI [9].

Due to GUI software being typically created using rapid prototyping, the GUI is constantly changing during the development, making maintenance of capture/replay or test scripts very expensive [10]. Therefore the current GUI testing techniques used in practice are incomplete, and a substantial amount of manual effort is required from the test designer [11]. A number of research results have shown MBT as a promising solution to overcome the maintenance weakness of capture and replay tools [12].

2.2. Model-Based Testing

Model-based testing (MBT) has been around for a relatively long time but has only recently begun to attract considerable interest in the domain of software testing. Generally in MBT, the SUT is represented by a model describing its expected behavior at a higher abstraction level, and a set of chosen algorithms are used to generate tests from this model [4]. MBT is expected to allow more adequate software testing because it is rooted in automated procedures, avoiding manual error prone activities [5].

The MBT process can be divided into five main steps: 1) modeling the SUT and / or its environment, 2) generating abstract test cases or sequences from the models, 3) concretizing the abstract tests into executable tests, 4) executing the tests on the SUT and assigning verdicts, and 5) analyzing the test results. Usually some kind of MBT tool is used to generate abstract test cases from a behavioral model of the SUT. Many MBT tools allow test engineers to control the focus and number of the test cases, and transforming the abstract tests into executable test scripts often requires some input from the test engineer [12].

Even though MBT is useful and effective, there are some challenges in adopting it into industrial software development. One generic barrier is its complexity, requiring deep expertise in formal methods. Also, MBT cannot be applied in every situation, for example when using legacy or 3rd party systems without behavior models or proper specifications [13].

Another reason is that a substantial amount of manual effort is needed to create the models for testing. One approach to provide automated assistance for model creation for testing purposes is observation-based modeling that reverses the traditional MBT approach of going from specification to implementation to going from implementation to specification [4]. Unfortunately, currently there is no tool support for observation-based GUI modeling.

2.3. Model-based GUI Testing

There are several approaches for modeling the GUI to automate some aspects of GUI testing. Using UML in MBT has been extensively studied, also for GUI testing as in [2]. However, the most popular models for GUI test case generation are different kinds of state machine models, such as finite-state machines (FSM) [11].

The key idea is that a test designer represents a GUI's behavior as a state machine where each input event may trigger an abstract state transition in the machine. A path in the state machine represents a test case and the abstract states of the state machine may be used to verify the concrete state of the GUI during test case execution. Since finite-state machines (FSMs) have been argued to have scaling problems for large GUIs, various extensions for FSMs, such as adding variables to the model or dividing the state space into several state machines, have been developed to reduce the number of abstract states. However, all of these techniques require substantial amount of manual effort, and none of them are commonly used in the industry [11]. To reduce the manual modeling effort, there are approaches to automate the creation of the GUI models for testing purposes, such as [11], [14] and [15].

In [11], GUITAR tool [7] is used to execute and reverse engineer the Java GUI application to generate models for testing purposes. First the GUITAR tool rips the Java GUI application and creates an XML file containing structural information of all windows, widgets, their attributes, and user events of the GUI. Then the created XML file is converted into an event-flow graph (EFG). The EFG is used to generate an event-interaction graph (EIG) and the mappings between EFG and EIG. These are then used to generate GUI test cases that are sequences of GUI events. The GUITAR tool also supports the execution of the generated test cases on the Java GUI application. One of the disadvantages of this approach and the GUITAR tool is that parts of the retrieved information may be incomplete or incorrect, and therefore major parts of the application under test (AUT) may be missing from the created model. When the created model is incomplete, it may be impossible to execute the generated test cases. Also, there is no support for manually refining the generated models.

[14] uses GUISurfer tool for parsing the source code of a Java or Haskell application to generate an Abstract Syntax Tree (AST). Then a combination of strategic programming and program slicing techniques is used to extract the GUI fragments from the AST. Based on analysis of the AST, GUISurfer is able to generate visual models, such as state machines and behavioral graphs. One downside of this approach is that the source code of the GUI application is required. Also, since the approach is based on the source code instead of executing the GUI applications, the run time aspects like timing and layout are not dealt with.

[15] introduces REGUI tool and a semi-automated approach to generate Spec# models from Windows GUI applications. User actions are required to guide the tool during the creation of the model, and the generated model has to be completed manually. The approach solves some problems related to the lack of valid input values while executing the GUI application and supports manual elaboration of the generated models. However, compared to other presented approaches, more effort is required in the creation of the models.

Our approach, presented in Chapter 3, aims to reduce the effort required to create the models, provide human readable graphical models that can be refined manually, and provide tool support for MBT of Java GUI applications. Also, the created graphical models of the AUT allow checking the implementation against the requirements of the system.

3. Automated Modeling and Testing of Java GUIs

This chapter presents our approach, the GUI Driver tool that we implemented as a proof-of-concept and a tool chain for automated modeling and testing of Java GUI applications. Our aim is to reduce the manual effort required to create models for GUI testing by providing tool support for automatically generating models suitable for MBT.

In our approach, the models are generated while automatically executing and observing the AUT. The generated models include structural tree models presenting the GUI components and their properties, and a GUI state model presenting the behavior of the GUI and mapping the structural models into the abstract states. Structural models are generated for each state of the GUI application and saved into XML files. Comparing two consecutive structural models makes it possible to observe the changes happening in the GUI while automatically interacting with the GUI application. The GUI state model represents the behavior of the GUI application as a state machine, presenting GUI states as nodes and interactions between the states as edges. The structural models are automatically mapped into the abstract states of the GUI state model.

For the GUI state model we provide also human readable graphical representation to allow checking the implementation against the requirements of the system. Normally, MBT is about manually modeling the SUT based on the requirements and automatically generating a test suite based on that model to check if the SUT fulfils the requirements. In our approach the generated models are based on the actual implementation, so the generated GUI state model should be compared with the requirements of the system. Another goal for the graphical modeling is to allow manual elaboration of the model, for example adding valid and invalid input values for text fields of the GUI. Also, automatically generated graphical models can help developers to understand and analyze an existing implementation if proper models of the system are missing. An example of the GUI state model is shown in Figure 1, and the same model is opened with yEd graph editor [16] in Figure 2.

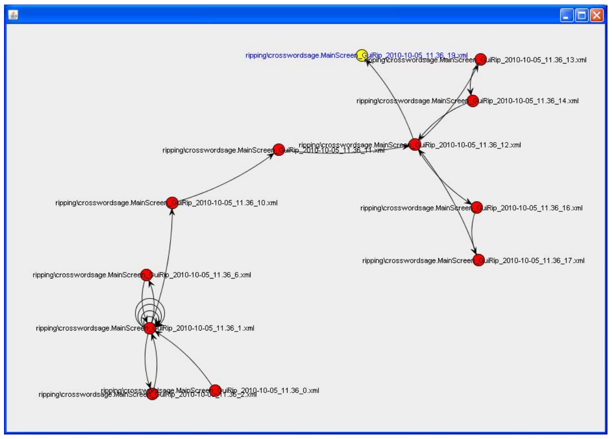

Figure 1. Generated GUI state model displayed by GUI Driver.

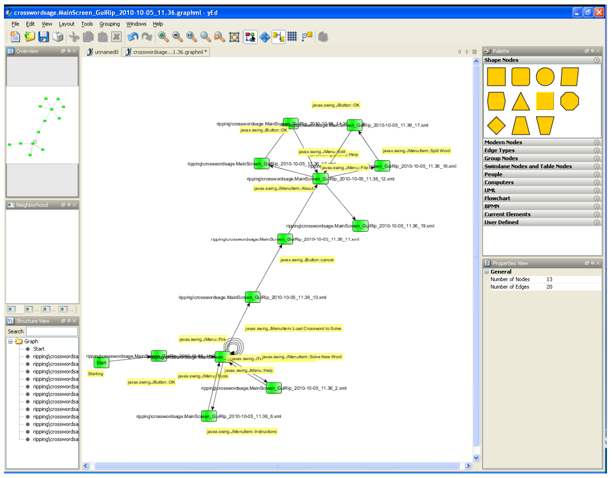

Figure 2. Generated GUI state model opened with yEd graph editor.

3.1. GUI Driver tool

As a proof-of-concept, we have implemented the GUI Driver tool to support our approach. It is a tool for programmatically driving a Java GUI application and generating models representing the states and behavior of the GUI. The created models are then used for MBT purposes.

In the high level architecture of the GUI Driver, presented in Figure 3, the main parts of the tool are GUI Driver, Model Generator and Test Executor.

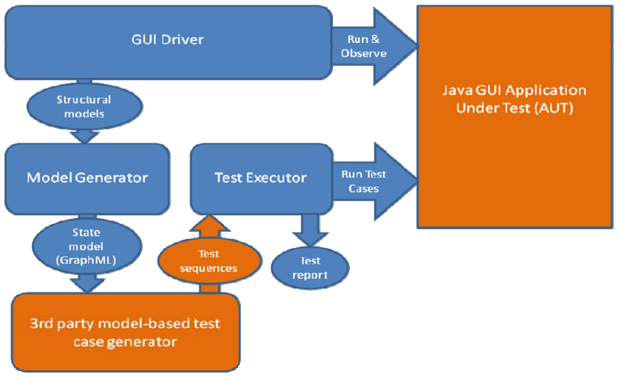

Figure 3. High level architecture of the GUI Driver.

After starting the AUT, the GUI Driver identifies the accessible and focused GUI window of the Java GUI application, creates a structural model of the window including the GUI components and their properties, and saves the model in an XML file. Then it identifies the enabled and visible interactions, selects and executes one of the available GUI actions and creates a new structural model after each executed GUI action. In case of selecting exit or close action during the execution, the AUT is restarted.

The Model Generator observes the behavior of the GUI, checks if a specific GUI state is a new one or one of the old states, and creates a state model of the GUI application suitable for MBT. Also, the generated structural models are mapped to the abstract states of the GUI state model. Currently, the GUI state model is saved in GraphML format that is applicable for the generation of test sequences with an open source MBT tool [17].

The Test Executor is able to parse test sequences generated by the MBT tool [17], execute the test sequences on the AUT, verify whether the expected states were reached after executing the specified GUI actions, and create a test report including information about executed GUI actions, reached GUI states, and abnormal behavior like unhandled exceptions.

3.1.1 Automatic selection of GUI actions. The GUI Driver uses certain rules and preferences for selecting one of the GUI actions provided by the AUT. The default way is to select one of the GUI actions that have not been selected in that GUI state earlier. If all of the actions of the state have been selected earlier, Dijkstra’s algorithm, provided by an open source Java library, JUNG [18], is used to find an action that leads to a state with unselected actions.

The enabled actions of consecutive states are compared and new GUI actions are preferred over the ones that were available in the previous state. That way after opening a drop-down menu it is probable that one of the actions of that menu is selected. If there are many evenly preferable actions, one is selected randomly.

3.1.2 Identifying the visited GUI states. After generating a structural model of the AUT, the GUI Driver compares the reached GUI state to the states visited earlier to check, whether the state is a new one or one of the already visited GUI states. The first criteria for a GUI state to be considered the same as one of the already visited states is that both states must have the same enabled GUI actions. The secondary check is to compare the structural models of the states. The number of the structural changes that are tolerated for similar GUI states can be specified in the settings.

3.2. Tool Chain for Automated Java GUI Testing

To form a tool chain to cover MBT of Java GUI applications from automated modeling until executing the generated test cases, we have combined the GUI Driver tool with an open source MBT tool [17] and a free graph editor tool, yEd [16].

The first step is to run the GUI Driver on AUT and generate the models of the GUI application. To run the GUI Driver, some settings, such as path and main class of the AUT, have to be set in the properties file. The GUI Driver explores and models as many GUI states as possible without knowing valid data values for the input fields, and saves the GUI state model in GraphML format.

If needed, the generated GraphML model can be edited using yEd [16] graph editor. If needed, the generated model can be manually refined by adding valid input values, and then used for test case generation. The support for re-running the GUI Driver with the manually elaborated model to improve and extend the model is currently an ongoing work. This would allow iterative process of automatic model generation and manual elaboration of the generated model. In addition, automatically extending an existing GUI model would provide better support for testing GUI applications during the development time, when the GUI is rapidly changing.

The next step is to use the MBT tool [17] to generate test sequences from the GraphML model. The MBT tool supports different kinds of generators and stop conditions for the test sequence generation. Using the selected generator and stop conditions, MBT tool generates a test sequence and save it into a text file.

The last part of this tool chain is to use the GUI Driver Test Executor to execute the generated test sequence. While executing the test sequence and observing the AUT, the Test Executor compares the state of the AUT to the structural models created earlier to see if the expected GUI state was reached or not. As the result, the GUI Driver generates a test report including information about executed GUI actions, reached GUI states, and abnormal behavior like unhandled exceptions.

The tool chain was evaluated by automatically modeling and testing several open source Java GUI applications, such as [19]. During the testing, errors and usability issues, such as a cancel button that required valid input in a text field, dialogs without a way to cancel or to go back to the previous screen, and exceptions that were printed to standard output but not otherwise handled, were found in the AUTs. During the testing many new ideas how to improve the GUI Driver tool were discovered. The future work is discussed in more detail in chapter 4.

4. Conclusion and Discussion

In this paper we presented an approach for automatic modeling of Java GUI applications for model-based testing (MBT) purposes, implemented proof-of-concept tool support for the approach, and combined the implemented GUI Driver tool with an open source MBT tool to form a tool chain for automated modeling and testing of Java GUI applications. Our approach aims to reduce the amount of manual effort required to model GUI applications to enable automated testing.

The strengths of our approach in comparison to the related work include automatically generating human-readable graphical models while requiring none or only a little manual effort. The graphical models provide test engineers with a way to manually elaborate the models, for example inserting valid input values for specific input fields of the GUI application, and allow comparison between models based on the actual implementation and requirements of the system.

The tool chain was used to automatically model and test several open source Java GUI applications, resulting in the discovery of several unknown errors and usability problems. The approach seems promising but there are also limitations. As our approach is based on running and observing existing software, the AUT must be an executable Java GUI application, and the models are based on the actual implementation instead of designed and expected features. Also, more work is needed on the GUI Driver, as the proof-of-concept implementation is not yet mature enough to be used in the software industry.

In future, we plan to improve the GUI Driver so that the generated models and reports would inform about the detected usability issues and include information about the changes that happened in the GUI after a specific interaction. The GUI Driver should indicate more clearly the states that should be manually elaborated in the model and support iterative modeling containing manual and automated phases. Also, we plan to extend the approach to be also usable on other kinds of GUI applications, such as desktop applications not implemented in Java and web applications.

5. References

[1] B. A. Myers, "UIMSs, Toolkits, Interface Builders", Human Computer Interaction Institute, Carnegie Mellon University, May 1996.

[2] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier, "Automation of GUI Testing Using a Model-driven Approach", Proceedings of the 2006 international workshop on Automation of software test, International Conference on Software Engineering, Shanghai, China, 2006, pp. 9-14.

[3] B. A. Myers, S. E. Hudson, and R. Pausch, "Past, present, and future of user interface software tools", ACM Transactions on Computer-Human Interaction (TOCHI), Volume 7, Issue 1 (March 2000), pp. 3-28

[4] T. Kanstrén, "A Framework for Observation-Based Modelling in Model-Based Testing", VTT Publications 727, Espoo, Finland, 2010.

[5] P. Santos-Neto, R. Resende, and C. Pâdua, "Requirements for information systems model-based testing". In Proceedings of the 2007 ACM symposium on Applied computing, pp. 1409-1415.

[6] M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Satama, "Towards Deploying Model-Based Testing with a Domain-Specific Modeling Approach", Proceedings of Testing: Academic & Industrial Conference (TAIC PART 2006), Windsor, UK, IEEE CS Press, 2006, pp. 81-89.

[7] http://sourceforge.net/projects/guitar/ (accessed 6th of October 2010)

[8] K. Li and M. Wu, "Effective GUI Test Automation: Developing an Automated GUI Testing Tool", SYBEX Inc., Alameda, CA, 2004.

[9] A. M. Memon, "Automatically repairing event sequence-based GUI test suites for regression testing", ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 18, Issue 2 (November 2008), Article No. 4

[10] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and Evolving GUI-Directed Test Scripts", Proceedings of the 2009 IEEE 31st International Conference on Software Engineering (ICSE 2009), pp. 408-418.

[11] A. M. Memon, "An event-flow model of GUI-based applications for testing", Software Testing, Verification & Reliability, Volume 17, Issue 3 (September 2007), pp. 137 - 157.

[12] M. Utting and B. Legeard, "Practical Model-Based Testing: A Tools Approach", Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, 2006.

[13] A. Bertolino, A. Polini, P. Inverardi, and H. Muccini, "Towards Anti-Model-based Testing", Proceedings of International Conference on Dependable Systems and Networks (DSN2004), Florence, 2004.

[14] J. C. Silva, C. C. Silva, R. D. Gonçalo, J. Saraiva, and J. C. Campos, "The GUISurfer tool: towards a language independent approach to reverse engineering GUI code", Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems, Berlin, 2010, Germany, pp. 181-186

[15] A. M. P. Grilo, A. C. R. Paiva, and J. P. Faria, "Reverse Engineering of GUI Models for Testing", 5th Iberian Conference on Information Systems and Technologies (CISTI), Santiago de Compostela, Spain, 2010

[16] http://www.yworks.com/en/products_yed_about.html (accessed 6th of October 2010)

[17] http://mbt.tigris.org/ (accessed 6th of October 2010)

[18] http://jung.sourceforge.net/ (accessed 6th of October 2010)

[19] http://sourceforge.net/projects/crosswordsage/ (accessed 7th of October)