Evolution of Automated Regression Testing of Software Systems Through the Graphical User Interface

Pekka Aho
Knowledge Intensive Products and Services
VTT Technical Research Centre of Finland
Oulu, Finland
email: pekka.aho@vtt.fi

Emil Alégroth
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
email: emil.alegroth@chalmers.se

Rafael A. P. Oliveira
Dept. of Computer Systems
University of São Paulo
São Carlos, Brazil
email: rpaes@icmc.usp.br

Tanja E. J. Vos
Centro de Mtodos de Produccin de Software (ProS)
Universidad Politecnica de Valencia
Valencia, Spain
email: tvos@pros.upv.es

***Abstract*—Increasing and more ubiquitous use of mobile and Web applications with graphical user interfaces (GUIs) places more stringent requirements on the software's quality. Automated testing is used to ensure the quality but testing through the software's GUI is challenging and therefore a research topic that has grown during the last decade. However, despite of the evolution of automated GUI-based testing methods and tools, its industrial adoption has been limited. In this paper, we present a synthesis of the evolution of GUI-based test automation and propose a classification for methods and tools for automated regression through the GUI.**

Keywords-Graphical user interface; automated GUI testing; software systems; classification; categorization; state-of-the-art.

I. INTRODUCTION

Increasing and more ubiquitous use of all kinds of mobile and Web applications with GUIs makes our daily lives dependent on the software functioning without errors, increasing the importance of assuring the correct and reliable behavior of software systems. Modern GUI-driven applications are often connected and consist of distributed back-end services and sub-systems. Additionally, the GUI is often the primary interface to access the software's functionality, which also makes it a natural interface for testing, and in some cases, the only means to perform end-to-end testing.

The widespread use of iterative and incremental processes and continuous integration practices in software development has shortened release cycles and limited the time available for testing in each release. This trend poses a challenge since manual GUI-based testing is tedious, laborious [1], and requires a lot of time. This implies that GUI-based test automation, especially for regression testing should be applied to get confidence in the quality of each release. However, from the point of view of continuous integration processes, GUI-based testing is often too slow to be run after each code commit, because the test automation tool has to wait for the GUI to react before executing the next action of the test sequence. Larger automated GUI testing suites are therefore run only a few times a day or overnight.

According to IEEE Standard Glossary of Software Engineering Terminology [2], regression testing is "Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements." As such, regression testing aims to verify that the behavior of the system under test (SUT) remains consistent after changes to the SUT. Thus, if the changes in SUT intentionally affect the SUT's behavior, the regression test cases usually have to be updated to correspond to the new behavior. Otherwise, the changes may have been unintentional and a regression fault was found.

There are various terms used for automated GUI testing or GUI-based testing, depending on the authors and the objectives of the testing. In our case, automated testing of software systems through GUI would be the most accurate, but the other terms are used as well. The main point is that we are not testing only the software related to the GUI, but using the GUI as an interface for testing the whole software, including also the possible back-end services. The approach is usually black-box, without access to the source code of the system and often without detailed knowledge on the architecture or implementation of the system, and system testing focusing on the functional requirements, features and behavior of the system. However, automated GUI-based testing provides opportunities also for non-functional testing, such as performance and robustness testing.

Automated testing of software systems through the GUI is challenging and has therefore become a popular research topic during the last decade. Despite of the evolution of automated GUI-based testing methods and tools, no large scale industrial adoption of state-of-the-art methods and tools has been seen, and capture and replay (C&R) tools remain being the most popular GUI testing approach in the software industry. However, C&R tools are associated with high maintenance costs implying a need for more cost-effective GUI-based testing.

In this paper, we summarize the evolution of automated testing of software systems through the GUI in Section II and propose a classification for the methods and tools in Section III. We present the related work in Section IV, and Conclusion and Discussion in Section V.

II. EVOLUTION OF AUTOMATED GUI TESTING

C&R, also called as record and replay (R&R), is one of the earliest and most widely used approaches for automating regression testing of GUI software. In C&R approaches, a test automation tool is used to capture the user's interactions with the SUT during manual use. The tool can then automatically replay the recorded sessions or sequences of interactions against different versions of the software, automating the test execution for regression testing. The modern, more advanced C&R tools capture also the behavior of the GUI software and are able to notice if the behavior of a later version changes compared to the recorded behavior. Usually, each test case is a session of manual interaction and has to be recorded separately.

In general, the C&R approaches are easy and intuitive to use, the tools are mature and widely used, and it is possible to get fast results, for example decreasing the manual effort for regression testing through GUI. There is a wide selection of both commercial and open source C&R tools for most of the widely used operating systems (OS) or platforms, such as SeleniumHQ [3] for Web applications, Appium [4] for mobile applications, and Squish [5] for a variety of different platforms, although all of them can be used for more than just C&R testing.

The obvious disadvantages of C&R is the amount of manual effort required to record the test cases, and even more importantly, the amount of manual effort required to maintain the test suites. Hence, whenever the software changes, the test cases related to the changed parts of the GUI have to be manually retested to be recorded again.

The next step in the evolution of automated GUI testing was using keywords and action words to present the GUI testing scripts on a higher level of abstraction. The goal was to make it easier to reuse parts of test cases to create new test cases and reduce the maintenance effort of test suites after changes in the GUI by providing a clear separation of concerns between business logic and the GUI navigation needed to implement the logic [6]. Although one could argue that the modern C&R tools are exploiting keywords to allow easier maintenance of test cases, purely keyword-based approaches for GUI testing have not been widely adopted by the industry.

When model-based testing (MBT) was introduced in the testing community, it was also adopted into automated GUI testing. In model-based GUI testing (MBGT), the GUI and its behavior is modeled in a higher level of abstraction, using a modeling language supported by the selected test generation tool. In traditional MBGT, the models are created manually, and the generated abstract test cases have to be mapped or transformed into a lower level of abstraction to get concrete executable test cases that can be automatically executed against the SUT. In addition to the effort required to create the models for MBGT, also considerable expertise on formal modeling is required. TEMA Toolset [7] is an example of using MBT for testing concurrency issues in smartphone applications through the GUI.

In recent years, model extraction, also called as model inference, specification mining, reverse engineering or GUI ripping, has been widely used to automatically extract GUI models for testing purposes. The earliest approaches used static analysis on the source code of GUI software, which had the drawback that it failed to capture the dynamic behavior of the GUI. In dynamic analysis, the behavior of the GUI is instead analyzed during runtime interaction with the SUT. Some tools using dynamic analysis for model extraction require a user to interact with the GUI, in a similar way to C&R tools, but more recent tools are able to simulate the end user, automatically interacting with the components or widgets of the GUI.

Most dynamic model extraction approaches, such as [8]-[12], use the following process to capture the GUI model: 1) Capture the current state of the GUI as a snapshot of the screen visible for the end user, 2) Update the behavioral model of the GUI if it is extracted, 3) Analyze the interactions that are available for the end user, 4) Select one of the interactions using a random or a more intelligent selection strategy, 5) Execute the selected action and wait for the GUI to update, 6) Repeat the process from step 1. There are small differences on what is considered as a state of the GUI, but usually it consists of the windows or screens visible to the user, the components or widgets of each of the screens, and properties and values of each of the widgets. If a behavioral GUI model is extracted as well, the differences between the approaches are more significant. Although there are a lot of publications around GUITAR [8] that uses event-based models, most approaches use finite state-machine (FSM) –based models and graphs to present the behavior of the GUI.

In most approaches, the GUI state, a snapshot of the visible GUI, is captured using some kind of application programming interface (API) provided by the OS, such as Windows Automation API [13], or a GUI library, such as Jemmy framework [14] for Java-based GUIs. The benefit of these APIs is that they provide the GUI information in a detailed and hierarchical way. The downside is that such APIs have not been standardized and in practice, model extraction tools have to implement support for several APIs and libraries to cover a wide variety of GUIs. Another option is Visual GUI Testing (VGT), using image recognition on partial images of the GUI and screen captures to extract the state of the GUI [15]. The benefit is the independence of the platform specific APIs and libraries, but the downside is that the visual approaches are not as accurate and detailed. In optimal cases, the model extraction tools are able to reach all parts of the GUI and extract an accurate behavioral model of the GUI. However, automatically extracting GUI models is still an active research topic.

The obvious restriction with the extracted models is that they are based on the observed behavior of the implementation, instead of expected or required behavior defined in requirement specifications. Therefore the extracted models are ill suited for conformance testing without manually elaborating the models before test case generation. However, the extracted models can be used for reference or regression testing. In optimal cases, using extracted models for generating test cases may achieve a high level of automation in regression testing through the GUI [8], but the quality and effectiveness of these approaches maybe lacking for software that is still under development. The challenge is that when the GUI changes, the automatically extracted reference model has to be extracted again, and then the test cases have to be generated based on the new reference model. As a result, the old test cases are failing and giving false positives for the changed parts of the GUI, and the new parts of the GUI are completely missing from the old test cases. Although the process of model extraction, test case generation and test execution can be fully automated, updating the reference model results a GUI version that has to be tested manually or using other means to ensure the correctness of the new reference model. Otherwise the newly generated test cases could use faulty behavior as their test oracle.

The latest step in the evolution of automated regression testing through GUI has been automated regression analysis based on comparison of automatically extracted GUI models [9]. This approach overcomes the problem of having to re-generate the test cases by not having test cases at all. Whenever the GUI changes, a new model is automatically extracted and compared to the previous version. All the changes are reported for the test engineer, and the manual work is limited to deciding if the change was intentional or a regression fault.

In addition, a lot of smaller scale evolution is studied in academia, improving the automated regression testing through the GUI. For example, automating the debugging of failures found during automated GUI testing [16] is definitely improving the level of automation of the whole software development process.

III. CLASSIFICATION FOR AUTOMATED TESTING OF SOFTWARE SYSTEMS THROUGH GUI

In this section, we propose a 2-axis classification of methods and tools for automated testing of software systems through the GUI, illustrated in Figure 1. Our intention is to provide a baseline for comparison between tools and methods for automated GUI-based testing, as a suitable public categorization is currently missing. Many of these methods and tools for GUI-based testing are still academic or proof-of-concept tools, but we hope that in the future, when the tools have matured and there is more tools to select from, this classification helps the industry in selecting the tools and methods suitable for their needs.

The vertical axis of our classification follows the three generation classification proposed by Alégroth et al. [15] but

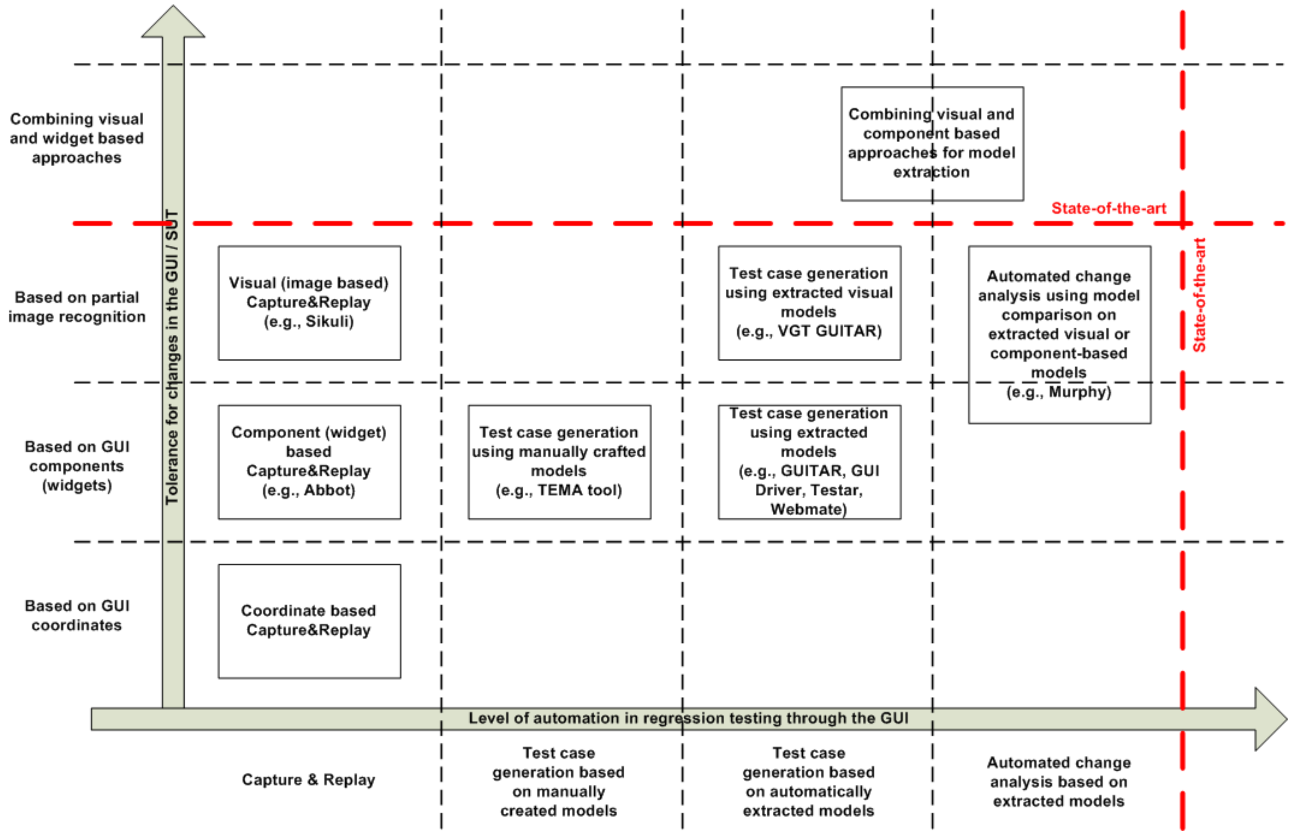
Combining visual and widget based approaches
Based on partial image recognition
Based on GUI components (widgets)
Based on GUI coordinates
Tolerance for changes in the GUI / SUT
Combining visual and component based approaches for model extraction
State-of-the-art
Visual (image based) Capture&Replay (e.g., Sikuli)
Test case generation using extracted visual models (e.g., VGT GUITAR)
Automated change analysis using model comparison on extracted visual or component-based models (e.g., Murphy)
State-of-the-art
Component (widget) based Capture&Replay (e.g., Abbot)
Test case generation using manually crafted models (e.g., TEMA tool)
Test case generation using extracted models (e.g., GUITAR, GUI Driver, Testar, Webmate)
Coordinate based Capture&Replay
Level of automation in regression testing through the GUI
Capture & Replay
Test case generation based on manually created models
Test case generation based on automatically extracted models
Automated change analysis based on extracted models

Figure 1. Classification for automated regression testing through GUI.

is named as tolerance for changes in the GUI. We have also included a fourth generation, not discussed by Alégroth et al., that combines the visual and API-based approaches, getting the benefits of both approaches, as discussed in Section II. As the combination would not bring any novel approaches to the field, we could call it 3.5th generation.

The earliest C&R approaches captured user actions in exact mouse coordinates [15]. These coordinate-based tools are categorized into lowest level of tolerance for changes in the GUI, because even small changes in the GUI, such as changing screen resolution or window's location on the screen, usually breaks the test case. None of the modern tools are relying solely on mouse coordinates anymore.

The second generation [15], or the second level of tolerance, consists of API-based approaches, based on components or widgets of the GUI. The advantage of using some kind of API provided by the OS or GUI library is that the recorded or modeled interactions are mapped to components or widgets of the GUI, instead of mouse coordinates, giving the second generation approaches better robustness against GUI changes [15]. In model extraction, the API-based approaches are more accurate than third generation visual approaches. Abbot [17] is an example of API-based C&R tool for Java GUI applications.

The third generation [15], or the third level of tolerance, approaches are based on VGT, using image recognition on partial images of the GUI and screen captures to interact and assert the correctness of the GUI. In some cases, VGT might be more tolerant to layout changes, but it is more dependent on the graphical representation of the GUI than API-based approaches. If the graphical icon of a button is redesigned, the related test cases have to be updated. An example of visual C&R tool is Sikuli [18].

The horizontal axis presents the level of automation in regression testing through the GUI. C&R approaches present the lowest level of automation, as manual effort is required both in recording of the test cases and in maintaining the test cases by re-recording test cases related to the changed parts of the GUI. The keywords or action words–based approaches would belong also to this first group.

The second level of automation is MBT using manually created models. This categorization addresses specifically regression testing, as manually created models may introduce a lot more benefits into other types of testing, such as testing if the GUI software conforms to the requirements specifications. The manually created models can provide a lot more information on the expected behavior, enabling better possibilities for generating test cases with meaningful test oracles. However, in regression testing, the effort required for manually creating the models is significant, and updating the models after changes in the GUI also requires some manual effort. Although TEMA tool [7] is not designed specifically for regression testing, it would fall into this category, and it is based on Android APIs to interact with the GUI.

The third level of automation is generating test cases based on automatically extracted models. In optimal cases, the level of automation with these approaches can be high. As described in Section II, the question is if the quality and efficiency of this testing is sufficient when the GUI changes, if the correctness of the model or the GUI is not assured with other means. API-based GUI model extraction and using the extracted models for test case generation has been a major topic in GUI model extraction and testing research during the last 15 years and there are a lot of academic tools available, such as GUITAR [19], GUI Driver [11], Testar [10], and Webmate [20], although Webmate has been commercialized. There is also more recent research and VGT GUITAR tool that is using visual approach in model extraction and test case generation [15].

The highest level of automation currently available is automated regression analysis using model comparison between automatically extracted models of the GUI software. With this approach, the manual effort remains in deciding if the reported GUI changes were intentional or regression faults. The only tool currently available in this category is open source Murphy tool [21].

IV. Related work

Since the evolution of automated GUI-based testing, presented in Section II, is a sort of state-of-the-art study, in this section we present related state-of-the-art studies in addition to related work on classifying automated GUI testing.

Kull [22] summarized the state-of-the-art on automated extraction of GUI models for the purpose of generating tests from the extracted models. The author raised the problem of not having meaningful test oracles as the main challenge in using extracted models for test automation.

Banerjee et al. [23] used systematic mapping to study 136 articles related to GUI testing to classify the nature of the articles, the aspects of GUI testing being investigated, the nature of evaluation being used, and to draw some conclusions based on the results. The authors conclude that more comparison is required between academic and industrial tools and techniques, and that commercial tools for MBGT are missing.

Aho et al. [24] presented an extensive state-of-the-art study on automated extraction of GUI models for testing. In addition to giving an extensive background on GUI testing and model extraction, the study summarized the work of most of the leading researchers and research groups related to using extracted GUI models for automated testing.

Alegroth et al. [15] have proposed to classify the existing GUI based testing approaches into three chronological generations. The first generation consists of C&R approaches capturing exact mouse coordinates. The obvious disadvantage, in addition to the general disadvantages of the C&R approaches, is the dependence on the screen resolution. If the same GUI software is executed on a different platform with a different screen resolution, the recorded test cases do not necessarily work.

The second generation consists of approaches based on components or widgets of the GUI and cover MBT approaches in addition to C&R. The advantage of using some kind of API provided by the OS or GUI library is that the recorded or modeled interactions are mapped to components or widgets of the GUI, instead of mouse coordinates, giving the second generation approaches better robustness against GUI changes [15].

The third generation approaches are based on VGT, using image recognition on partial images of the GUI and screen captures to interact and assert the correctness of the GUI [15]. There are also C&R tools, such as Sikuli [18], that fall into this category. In some cases, VGT might be more tolerant to layout changes, but it is more dependent on the graphical representation of the GUI. If the graphical icon of a button is redesigned, the related test cases have to be updated.

This mainly chronological classification [15] is not sufficient, as it does not address the level of automation at all, and all three generations have also C&R approaches. The most common inducement for adopting test automation is reducing the manual effort and time required for testing. Therefore the level of automation or amount of manual effort has to be considered when evaluation test automation methods and tools. Hence, in Section III we have proposed an improved classification of methods and tools for automated GUI testing having a second axis for the level of automation.

V. CONCLUSIONS, DISCUSSION AND FUTURE WORK

In this paper we summarized the evolution of GUI-based test automation and proposed a classification of methods and tools for automated regression testing through the GUI.

The classification proposed in this paper does not take into account all aspects of test automation related to testing through the GUI. Instead, it focuses on regression testing. Our intention is to provide a baseline for comparison between different tools and methods, and we hope that in the future the classification helps the industry in selecting the tools and methods most suitable for their needs. The variety of available test automation tools is growing, and it will become more challenging to select the tools that are best suited for the needs of a specific project.

Based on the state-of-the-art study, in the future we plan to address the lack of performance of GUI model extraction by executing the GUI being modeled in several virtual machines in parallel. Hence, we hope to get the automated regression analysis to be fast enough for the expectations of continuous integration processes. The same functionality could be used more generally to make automated UI test execution faster. We plan to work on combining component or API-based approach with visual image recognition aspects to make UI model extraction more accurate and tolerant for changes in the UI. Another future research subject would be using static analysis on the source code of the UI application to extract possible input combinations for increasing the coverage of model extraction.

REFERENCES

[1] M. Grechanik, Q. Xie, and C. Fu, “Creating GUI Testing Tools Using Accessibility Technologies,” in International Conference on Software Testing Verification and Validation Workshops (ICSTW’09), 1-4 April 2009, Denver, CO, USA, pp. 243-250, IEEE.

[2] IEEE Standard Glossary of Software Engineering Terminology, 1990, IEEE, New York, USA.

[3] SeleniumHQ, a tool for automating Web testing, http://www.seleniumhq.org/ [retrieved: Apr, 2016]

[4] Appium, a tool for automating mobile app testing, http://appium.io/ [retrieved: Apr, 2016]

[5] Squish GUI Tester, http://www.froglogic.com/squish/gui-testing/ [retrieved: Apr, 2016]

[6] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara, ”Model-Based Testing Through a GUI,” in Proceedings of the 5th international conference on Formal Approaches to Software Testing (FATES'05), July 11, 2005, Edinburgh, UK, pp. 16-31.

[7] TEMA Toolset, an MBT tool for mobile GUI testing, http://tema.cs.tut.fi/ [retrieved: Apr, 2016]

[8] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, “GUITAR: an innovative tool for automated testing of GUI-driven software,” in Automated Software Engineering, March 2014, Volume 21, Issue 1, pp 65-105, Springer.

[9] P. Aho, M. Suarez, T. Kanstrén, and A. Memon, “Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing,” in Proceedings of 2014 IEEE International Conference on Software Testing, Verification, and Validation Workshops, 1-4 April 2014, Cleveland, OH, USA, pp. 343-348.

[10] T. Vos, P. Kruse, N. Condori-Fernández, S. Bauersfeld, and J. Wegener, “TESTAR: Tool Support for Test Automation at the User Interface Level,” in International Journal of Information System Modeling and Design, IJISMD 2015, vol.6 (3), July-September 2015, pp. 46-83.

[11] P. Aho, N. Menz, T. Räty, and I. Schieferdecker, "Automated Java GUI Modeling for Model-Based Testing Purposes,” in Proc. 8th Int. Conf. on Information Technology: New Generations (ITNG), 11-13 Apr 2011, Las Vegas, USA, pp. 268-273.

[12] Y. Miao and X. Yang, "An FSM based GUI test automation model," in Proc. 2010 11th Int. Conf. on Control, Automation, Robotics & Vision (ICARCV), Singapore, 7-10 Dec 2010, pp. 120-126.

[13] Windows Automation API, an interface for Windows GUI automation, https://msdn.microsoft.com/en-us/library/windows/desktop/ff486375(v=vs.85).aspx [retrieved: Apr, 2016]

[14] Jemmy framework, GUI library for Java GUI automation, https://jemmy.java.net/ [retrieved: Apr, 2016]

[15] E. Alégroth, Z. Gao, R. Oliveira, and A. Memon, “Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: an Empirical Study,” in 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 13-17 April 2015, Graz, Austria, pp. 1-10, IEEE.

[16] E. Elsaka and A. Memon, “A Fully Automated Approach for Debugging GUI Applications,” in 1st International Workshop on User Interface Test Automation (INTUITEST 2015), 19 October 2015, Sophia Antipolis, France, pp. 51-60.

[17] Abbot Java GUI Test Framework, http://abbot.sourceforge.net/ [retrieved: Apr, 2016]

[18] Sikuli Script, a visual GUI automation tool, http://www.sikuli.org/ [retrieved: Apr, 2016]

[19] GUITAR, a GUI Testing Framework, https://sourceforge.net/projects/guitar/ [retrieved: Apr, 2016]

[20] Webmate, fully automated web testing tool, https://webmate.io/ [retrieved: Apr, 2016]

[21] Murphy, an open source GUI model extraction, change analysis and testing tool, https://github.com/F-Secure/murphy [retrieved: Apr, 2016]

[22] A. Kull, “Automatic GUI Model Generation: State of the Art,” 2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops, 27 Nov - 30 Nov 2012, Dallas, TX, USA, pp. 207-212.

[23] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, “Graphical user interface (GUI) testing: Systematic mapping and repository,” in

Information and Software Technology, Volume 55, Issue 10, October 2013, pp. 1679-1694.

[24] P. Aho, T. Kanstrén, T, Räty, and J. Röning, "Automated Extraction of GUI Models for Testing," in Advances in Computers, Vol. 95, 2014, pp. 49-112, Burlington: Academic Press.