2015 12th International Conference on Information Technology - New Generations

Making GUI Testing Practical: Bridging the Gaps

Pekka Aho
VTT Technical Research Centre of Finland
Oulu, Finland
pekka.aho@vtt.fi

Matias Suarez
F-Secure Ltd
Helsinki, Finland
matias.suarez@f-secure.com

Atif Memon
University of Maryland
College Park, MD, USA
atif@cs.umd.edu

Teemu Kanstrén
VTT Technical Research Centre of Finland
Oulu, Finland
teemu.kanstren@vtt.fi

***Abstract*—The effort and expertise required for manually crafting the models for model-based testing (MBT) is a major obstacle slowing down its industrial adoption. For implemented and executable systems, there are approaches to automate some part or even the whole process of creating the models for MBT. Recently, using extracted models for testing graphical user interface (GUI) applications has been a popular area of research, but most of the proposed approaches have limitations and restrictions on what can be modeled, and the software industry has not adopted these approaches. In this paper, we try to identify the gaps between the academic approaches and tools and industrial requirements hindering the industrial adoption, and try to suggest practical solutions to the identified gaps.**

***Keywords*—Graphical User Interface; GUI test automation; reverse engineering; model extraction; model-based testing.**

I. Introduction

A significant part of our daily lives is dependent on the reliability and quality of software. The software industry is facing the challenge of constructing more and more complicated and large systems with lower budget and less time to deliver. Unfortunately, testing is often the first software development activity to feel the budget cuts and closer delivery dates. Practically all end user applications have a graphical user interface (GUI) and the size and complexity of modern GUIs is also increasing [1]. The software industry is trying to address these challenges by increasing the use of test automation.

Model-based testing (MBT) has been a popular area of research for a long time [2], but its adoption is extremely dependent on tool support. The tools have only recently matured to a level comfortable for the industry to adopt MBT into larger scale use. In addition to the tools, the main challenges in industrial adoption of MBT are the specialized expertise and a considerable amount of effort required for creating the formal models [3] and the mapping between the model and the actual system required for generating executable test case [4]. When an implemented and executable system is being modeled, there are various approaches to automate some part or even the whole process of creating the models for MBT. Recently, especially GUI software has been a popular area of model extraction and testing research. Unfortunately most of these approaches have limitations and restrictions on the GUI applications that can be modeled, and the industry adoption has been very limited.

In this paper, we try to identify the gaps between the academic methods and tools and industrial requirements that are hindering the adoption of model extraction for automated GUI testing. The gaps were collected both from the published research results and from experiences of industrial companies during joint research projects on test automation in Europe. We present the identified gaps and try to suggest practical solutions to each of them. We identified gaps both in automatically extracting the GUI models and in utilizing the extracted models in testing:

- Gap 1 (G1): Scaling up to non-trivial systems while maintaining sufficient accuracy in extracted models.
- G2: Reaching a sufficient coverage in a reasonable time for model extraction.
- G3: Validating the correctness and coverage of the extracted models (e.g., comparing to the expected behavior).
- G4: General applicability of the provided tools (e.g., limitations and restrictions on the systems being modeled).
- G5: The introduction and adoption effort (e.g., learning curve, interoperability with existing tools and processes).
- G6: Minimizing the manual effort in GUI testing.
- G7: Minimizing the maintenance effort (e.g., from GUI changes).

In our previous work, we have introduced our platform independent approach for automatically extracting models of GUI applications [5] and presented our experiences on using Murphy open source tools [6] for automated modeling and testing of commercial GUI applications [7]. In this paper, we analyze the problem domain from a wider perspective, trying to identify the challenges and generalize possible solutions.

This work was partially supported by grant number CNS-1205501 by the US National Science Foundation, and a part of ITEA2/ATAC and COIN projects funded by the Finnish Funding Agency for Technology and Innovation TEKES.

II. Background and Related Work

A. Automating the Construction of GUI Models for Testing

There are various types of models used for model-based GUI testing (MBGT), the most popular being state-based models. The key idea is that the behavior of a GUI application is presented as a state machine, nodes of the model are GUI states, edges are events and interactions, and each input event may trigger an abstract state transition in the machine. A path of nodes and edges in the state machine, i.e., sequence of states and state transitions in the GUI, represents a test case. The abstract states of a state machine are used to verify the concrete states of the corresponding GUI application during the test case execution [1]. Reverse engineered state-based models are used for testing GUI applications in various approaches, e.g., GUI Driver [8] and GuiTam [9] for Java GUI applications, Crawljax [10] and DynaRIA [11] for rich internet applications (RIAs), and AndroidRipper [12] for Android applications.

Another popular format for extracted GUI models for testing is event-based models. Memon's team has implemented GUITAR [13], a model-based system for automated GUI testing, to execute and observe GUI applications for automatically constructing event-based models that are used for MBGT. Memon et al. [14] present DART, a framework for using automatically crafted GUI models for re-testing the modeled GUI applications, e.g., smoke testing nightly or daily builds of GUI software. Xie et al. [15] introduces rapid crash testing and defines a tighter, fully automatic GUI testing cycle for rapidly evolving GUI applications. The key idea is to test the GUI each time it is modified, i.e., at each code commit.

State space explosion is a challenge in modeling any non-trivial system, especially when using state-based models. So far the other modeling approaches, e.g., event-based models [1], have not provided solutions for this challenge. Any nontrivial program has a large number of possible states, depending on the definition of the state and how to distinguish them. Over the years, optimizations to the original learning algorithms have yielded significant improvements in terms of the speed of model inference and the size of extracted models, making it possible to infer state space sizes of 100,000 states or more, which is sufficient to test many kinds of industrial applications [16]. The challenge is to find the balance between increasing expressiveness to extract more accurate models and keeping the computational complexity on feasible level for model inference and model checking. Abstracting away too much information from the SUT increases the risk of losing opportunities to discover faults [16]. In most proposed GUI model extraction approaches, the modeled applications have been rather small, not showing that the model extraction methods scale up to non-trivial GUI applications (Gap 1).

In general, a challenge in any specification mining approach aiming to use the extracted models for testing is making the approach cost-effective in terms of its adoption [17]. For GUI software, there are existing frameworks that provide the instrumentation for observing the GUI, such as Jemmy [18] and Microsoft UI Automation [19], and the technical domain of GUI applications is wide but similar enough to be cost-effective through re-using the same expertise and tools on various systems. Nevertheless, most GUI model extraction approaches and tools have limitations and restrictions on the systems that can be modeled (Gap 4), e.g., modeling only GUIs implemented in a specific programming language or on a specific platform, such as AJAX [20], Java [21], [22], [23] or Android [12], [24], [25].

Most of the recent GUI model extraction approaches are based on dynamic reverse engineering, i.e., executing the application and observing the runtime behavior of the GUI. A major challenge in automatically traversing or crawling through the GUI is providing meaningful input for the input fields of the GUI, such as providing valid username and password for a login screen, without predefined instructions from the user [8]. Usually, some human intervention is required during the modeling process to achieve a good coverage with dynamically reverse engineered models [8], meaning that the modeling is assisted manually by a person during the reverse engineering process, or the automatically generated initial model is reviewed, corrected, and extended manually by a person after the model extraction [26]. The efficiency of these semi-automatic modeling techniques depends on the degree of required human intervention [26]. Although semi-automated processes of providing input values [8] have been proposed, most automated approaches are not able to reach all parts of the GUI during model extraction (Gap 2).

When extracting models automatically by observing an existing system, the generated models are based on the observed implementation. As such, the generated models include also the undesirable behavior of the system, instead of capturing the requirements or expectations of the system [27]. This limits the possibilities in utilizing of the models. Most model extraction approaches have not addressed the challenge of validating the correctness and sufficient coverage of the modeled behavior (Gap 3).

B. Utilizing Generated Models in GUI Testing

Extracted models are based on observed behavior of an implemented system, instead of requirement specifications or expected behavior. Therefore, without elaboration, the extracted models are not well suited for generating test cases and test oracles, as in traditional MBT. Conformance testing, i.e., testing the implemented system against its specifications, requires a link to the requirements before using the extracted models for test case or test oracle generation [27].

The challenge in automated GUI testing, especially when using automatically extracted GUI models for testing, is to provide meaningful test oracle information to determine whether a test case passes or fails [28]. With conventional software, a test case usually consists of a single set of inputs, and the expected result is the output that results from completely processing that input, and the oracle is invoked when the actual observed output is compared with the oracle's expected output after executing the test case [28].

With GUI testing, the input may consist of a long sequence of actions, and there is no specific output as each executed action may affect the state of the GUI. In MBGT, the oracle information consists of a set of observed properties of all the windows and widgets of the GUI [29]. The execution outcome may depend on the internal state of the GUI application, the state of other entities (objects, event handlers) and the external environment, and may lead to a change in the state of the GUI or other entities. Moreover, the outcome of an event's execution may vary based on the sequence of preceding events or interactions seen thus far [30].

An incorrect GUI state during a test sequence can lead to an unexpected screen, making further test case execution useless or impossible [28]. Therefore, the correct state of the GUI has to be verified after each execution step during a test case, interleaving the oracle invocation with the GUI test case execution [28]. Otherwise detecting the actual cause of an error can become difficult, especially when the final output is correct but the intermediate outputs have been incorrect [28].

The oracle information for automated GUI testing may be selected or created either automatically or manually [31] based on requirements or other formal specifications of the GUI or observed behavior of an earlier, presumably correct version of the software [32]. By varying the level of detail of oracle information and changing the oracle procedure, a test designer can create different types of test oracles, depending on the goals of the specific testing process used [32]. The different types of test oracles have a significant effect on test effectiveness and cost of testing [32].

In most approaches that use extracted GUI models for testing, the test oracles are based on the observed behavior of an earlier version of the GUI application. Using this kind of test oracles, in literature often called reference testing, changes and inconsistent behavior of the GUI can be detected and the models can be used for automated regression testing, but conformance testing is problematic [26]. However, some defects, such as crashes and unhandled exceptions, can be detected without the use of application specific test oracles [33], making it possible to begin the testing of the GUI application already during the dynamic reverse engineering process, as in [8].

Most of these academic approaches and tools have been validated by modeling and testing open source applications and simple proof of concepts, and the adoption by the industry has been very limited. In [7] we shared our experiences from a long term industrial evaluation of Murphy tools, showing that model extraction techniques can be utilized on non-trivial commercial GUI applications and the extracted models can be successfully used to automate and support various GUI testing activities in software industry.

III. Bridging the Gaps in GUI Testing

In this Section, we present the gaps we have identified to hinder the industrial adoption of the state-of-the-art academic approaches on automated modeling and testing of GUI applications. The gaps were collected both from the published research results and from experiences of industrial companies during joint research projects on test automation in Europe. We propose practical solutions to each of the identified gaps. Gaps G1 - G4 are related to automatically extracting GUI models for testing and G5 - G7 are related to using the extracted models to automate and support GUI testing.

A. *G1: Scaling Up to Non-Trivial Systems While Maintaining Sufficient Accuracy in Extracted Models*

Despite of the evolvement of hardware and algorithms used in model extraction, state space explosion remains as a challenge in creating state-based models of any non-trivial system. The challenge is to find the balance between increasing expressiveness to extract more accurate models and keeping the computational complexity on feasible level for model inference and model checking. Abstracting away too much information from the SUT increases the risk of losing opportunities to discover faults [16].

A GUI state comprises of a set of objects and their property values and any difference in number of objects or property values may mean a different state. Some property values may have huge or even infinite number of possible values, which in turn makes the number of GUI states huge or infinite. Without a proper method to limit the explosion of GUI states, it is infeasible to utilize model-based GUI testing methods [33]. The challenge is to find the balance between increasing expressiveness to extract more accurate models and keeping the models small enough to be computationally feasible for model inference and model checking [16]. Abstracting away too much information from the system under test increases the risk of losing opportunities to discover faults [16] and losing context information of the events and interactions, i.e., having ambiguous states or state transitions in the model.

The solution for reducing the states of the model is, of course, abstracting or ignoring some of the properties or values of the GUI when distinguishing the states of the model, but the challenge is how to find the right level of abstraction and automatically choose the important properties and values. An efficient solution for reducing the number of GUI states is ignoring the data values, such as text on input fields, and concentrating on the interactions that are available for the end user in each GUI state. To capture the context of executed interactions, the data values can be saved into the properties of the state transitions, as in [8]. The downside is that the reduction in states will result increased amount of possible transitions.

Murphy tools [7] use parameterized screenshots of the GUI in a similar way to abstract away the data values from the GUI states. The proposed level of abstraction performed well against the state space explosion. The size of the modeled GUI applications was of the order of magnitude of hundreds of thousands to millions lines of code, and the size of the extracted models was between 81 – 178 nodes (GUI states), which is still at a computationally feasible level. Even though the graphical presentation of the extracted state model does not show all the captured information, internally Murphy captures also the preceding actions as context information of the GUI states and transitions. In case of ambiguous state transitions, Murphy adds another transition with the same action into the graphical presentation, for example there could be transitions 'Ok-button' and 'Ok-button (1)' leading to two different states, depending on the data values of the GUI, and the context would be captured in the internal model.

Another important aspect in state reduction, and more generally in model extraction, is filtering out the external changes that might affect the detailed behavior of the system, but are not relevant for the modeling purposes. A practical solution is to use virtual machines to stabilize the model extraction environment. A new, clean virtual machine can be launched automatically each time the GUI application has to be started or restarted.

B. *G2: Reaching a Sufficient Coverage in a Reasonable Time for Model Extraction*

When using some sort of GUI automation for automatically exploring or crawling through the GUI during automated model extraction, the challenge is how to access all parts of the GUI to have a good coverage in the extracted models. For example, it is very improbable to find matching username and password to get beyond a login screen with random generation algorithms, if the user has not provided any predefined set of test data []. Random input generation can be used to improve the coverage of extracted models, but finding specific values with random methods requires too much time, slowing down the model extraction process. When using extracted models for testing, the parts of the GUI that are missing from the models will not be covered with the test cases automatically derived from the models. Usually, the user, e.g., test engineer, has to provide valid combinations of input before or during the model extraction process. The amount of manual effort should be minimized by providing practical tool support for the user. Another option could be using static analysis of the source code to generate meaningful input, but for example authentication data, such as usernames and passwords, are usually stored in databases instead of the source code of the system.

If the user has to provide the valid input combinations during the model extraction process, a practical way is to use the actual GUI of the modeled application, as in [8]. However, it might be easier to provide input for multiple states of the GUI by using a visual presentation of the extracted model, so that the user can select the state and then the widgets for the input. That way, the user does not have to monitor the progress of the model extraction process as often, but preserving the manually provided data when re-generating the models remains challenging. With Murphy tools [7] all the application specific data and instructions for model extraction are stored in a script that is used to start the automated model extraction process. In practice, an iterative process is used to define and improve the scripts to extract models with sufficient coverage.

With unlimited time for model extraction, even the random methods will cover all parts of the GUI. In practice, the development and testing process will dictate the maximum time available for model extraction. It might be for example 10 hours if once per day model extraction executed over-night is sufficient, or a few hours if new model is automatically extracted 3 times a day, as in [7]. There are various ways to reduce the time required for model extraction. A common goal is to maximize the coverage, e.g., the number of GUI states covered, while minimizing the extraction time, e.g., the number of transitions required. One proposed solution is to use various extraction strategies based on classification of GUI widgets, as in [34], to select the interactions with highest probability to result new GUI states.

Another factor to consider is the manual effort required for reaching the automated model extraction. For example, to define a model extraction script for Murphy tools [7] covering all the possible states and transitions of a complex GUI application would require a lot of time and effort, even though the model extraction after that would not require any manual assistance or guidance. Some of the GUI flows are difficult to explore, for example dialogs shown only when the network connection is lost, and would provide only a little added value for the automated testing. Therefore, it is up to the test engineers to decide when a sufficient coverage, e.g., 80% of all the possible GUI flows, has been reached and the iterative process of improving the invocation scripts is finished. Of course, the duration of model extraction process also depends on the size and complexity of the GUI being modeled.

C. *G3: Validating the Correctness and Coverage of the Extracted Models*

With dynamic reverse engineering approaches, the extracted models are based on the observed behavior of the implemented system, rather than the expected behavior defined in requirements or other specifications. Therefore it is challenging to use them for automatically generating meaningful test oracles without manual elaboration [8]. Some approaches use extracted "as is" models for automating various testing activities, but usually the generated models have to be manually inspected or elaborated, validating the correctness or adding the expectations and requirements into the automatically extracted models. Generally, the goal of test automation is to reduce and avoid manual steps, and model extraction approaches should provide practical means to manually validate the correctness of the models, or to manually elaborate the generated models and preserve the manual changes when re-generating the models. If the behavior captured in the extracted models is validated against the specifications, it is possible to use the models for conformance testing, in addition to reference or regression testing.

The most practical solution to validate the extracted GUI models seems to be visual inspection [7]. The extracted models are illustrated in a high level of abstraction and the states and transitions of the model are visualized with screenshots of the actual GUI, so that the correctness of the model and the behavior of the modeled GUI application can be visually inspected and validated by the user, e.g., test engineer or UI designer, based on requirements, design, or other specifications. If the extracted model does not include all the parts or behavior of the GUI, the model extraction has to be improved (related to G2) by instructing the extraction tool to include the missing parts. If the model includes parts that are not in the specifications, and the model extraction worked correctly, the problem is either in the modeled GUI application or the specifications. Either the incorrect behavior of the application has to be fixed or the specifications have to be updated by discussing with the stakeholders.

If the UI design would be available in a standard machine-readable format, the validation of the extracted model could be done automatically, showing only the deviations in behavior for the user. Although such a format would be technically fairly easy to construct, the challenge would be getting all the UI designers to use this standard format.

D. G4: General Applicability of the Provided Tools

The problem with most model extraction approaches for GUI applications is that they limit the applications that can be modeled to specific programming languages or platforms, usually based on the instrumentation framework used for observing the GUI. It is challenging to provide platform independent GUI reverse engineering techniques and usually implementing support for each programming language and platform requires too much effort.

As the instrumentation provided by the GUI frameworks usually allows more detailed analysis of the GUI, we recommend using them to provide support for the most common GUI platforms. For the platform independent GUI analysis, automatically capturing and comparing screenshot images before and after each interaction for locating and analyzing the changes in the screen to determine the elements and behavior of the GUI seems to be the most efficient approach [5]. Comparing the images before and after launching the GUI application can be used to find the right GUI window to analyze. GUI elements that can be interacted with can be automatically detected and located for example by automating the use of 'tab' key to cycle through the focusable elements and comparing the automatically taken screenshots to find the changing areas, such as the bounding rectangle of the selected element that has the focus on the screen. The structure and behavior of the GUI can be analyzed from the screenshot images based on the clues that the GUI application or the platform (operating system) offers for the end user, such as the shape of the mouse cursor. The correct instrumentation framework to be used in the model extraction can be automatically selected or combined with platform independent analysis, or manually selected.

E. G5: The Introduction and Adoption Effort

A major obstacle hindering the adoption of new software engineering methods and tools is the effort required for introducing and adopting them into practical use. Small development organizations might be willing to change their whole development and testing environment to adopt new more efficient methods and tools into use, but most of the industrial companies require that the new tools can be integrated into the existing processes and tools. Therefore, when selecting or developing the methods and tools for automated model extraction and testing of GUI applications, integration into the existing development and testing environment should be taken into account. Another important factor is the learning curve, e.g., if using the method or tool requires a lot of training.

To give some guidelines to help in the integration, the generated models could be transformed into a format used by the existing MBT tools, and the generated test cases and test scripts should be automatically executable with the existing test automation tools and test environments, considering also virtualization, continuous integration, test reporting and bug tracking. When using continuous integration, a good regression testing practice is to automatically run the model extraction and model comparison scripts several times a day, and send warnings when changes in the GUI behavior are detected.

F. G6: Minimizing the Manual Effort in GUI Testing

Usually, it is not possible or feasible to automate all GUI testing. There are proposed approaches to automate even usability testing, but in practice, at least some test cases have to be executed manually or supervised to validate the user experience. The challenge is how to support manual GUI testing and minimize the amount of manual effort during activities that are too expensive or difficult to automate.

With an automated virtualization of the test environment, it is possible to provide model-based support for manual GUI testing. By providing a user interface with a graphical presentation of the extracted GUI model and letting the user to select a path through the model, it is possible to automatically execute the selected path while the user is watching and evaluating the user experience. Another option is to let the user to select a path or a state from the GUI model, and automatically launch the application and execute it into the selected state, so that the user can start the manual testing without the initialization effort, i.e., manually executing the test steps that are not interesting in the test case.

Using the Murphy tools [7] to support the execution of manual GUI test cases significantly reduced the time required for GUI testing. Although the reduction varied depending on the application being tested and the particular test cases, generally the results were very promising. For example, the manual execution of a GUI test case required over 30 minutes, and less than 10 minutes was required to test the same test cases with the help of the extracted models and the Murphy tools. The main advantage was that the Murphy tool executed automatically the tedious and repetitive steps and the steps that required waiting time, leaving only the steps that required manual analysis and verification of the results for the user.

G. G7: Minimizing the Maintenance Effort

The observed behavior of an earlier version can be used as a test oracle to detect changes in the GUI during regression testing. Finding the changes between the versions is fairly easy, for example by comparing the structural models of the same GUI state of the two versions, as in [22]. Usually, the behavior of the GUI changes often during the development of the application due to the new features or improvements to the user experience. With automated regression testing, the test engineer has to go through the detected changes and decide if they were new features or incorrect behavior. In the case of new features the failed test cases have to be updated. Again, the challenge for test automation is how to support the user and minimize the manual effort.

To reduce the maintenance effort of regression testing, we propose using model comparison instead of test cases derived from the model. If the model is extracted automatically for each new version and there are no test cases, the maintenance effort is minimized. The test engineer still has to decide if the detected changes are new features or bugs, but that can be supported with tools. To help the user to decide if the change was intentional or incorrect behavior, an efficient way is to show the screenshot images of each state with detected changes and visualize the differences between the old and new version.

If the changes are detected by comparing automatically captured screenshot images, to reduce false positives, the image comparison algorithms can be made more tolerant to minor, irrelevant changes for example by transforming the screenshots into grayscale images and tolerating a small percentage of changes between the images.

IV. Discussion and Conclusion

In this paper we identified and presented several gaps between the academic state-of-the-art approaches and the requirements and practices of the industry on automated extraction of GUI models for testing. We tried to propose practical solutions to each of the presented challenges, but a lot of work remains in making the adoption of the methods and tools easier for the industry. We hope this paper will encourage the industry to evaluate and adopt techniques to automatically extract GUI models and utilize them in testing.

References

[1] A. M. Memon, "An event-flow model of GUI-based applications for testing", Software Testing, Verification & Reliability, Volume 17, No. 3, Sep 2007, pp. 137-157, John Wiley and Sons Ltd. Chichester, UK.

[2] M. Utting and B. Legeard, "Practical model-based testing: a tools approach", Morgan Kaufmann Publishers, San Francisco, USA, 2006.

[3] Holzmann, G. Z. and Smith, M. H. 2002. An Automated Verification Method for Distributed Systems Software Based on Model Extraction. IEEE Trans. on Software Engineering. 28, 4 (Apr. 2002), 364-377.

[4] A.M.P. Grilo, A.C.R. Paiva, and J.P. Faria, "Reverse engineering of GUI models for testing", 2010 5th Iberian Conf. on Information Systems and Tech. (CISTI), 16-19 Jun 2010, Santiago de Compostela, Spain, pp. 1-6.

[5] P. Aho, M. Suarez, T. Kanstren, and A.M. Memon, "Industrial adoption of automatically extracted GUI models for testing", Int. Workshop on Experiences and Empirical Studies in Software Modelling (EESSMod), 1 Oct 2013, Miami, Florida, USA, pp. 49-54.

[6] Murphy, open source tools for automated modeling and testing of GUI applications, https://github.com/F-Secure/murphy

[7] P. Aho, M. Suarez, T. Kanstren, and A.M. Memon, "Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing", Testing: Academic & Industrial Conference - Practice and Research Techniques (TAIC PART), 4 Apr 2014, Cleveland, OH, USA.

[8] P. Aho, N. Menz, and T. Räty, "Enhancing generated Java GUI models with valid test data", IEEE Conf. on Open Systems (ICOS), 25-28 Sep 2011, Langawi, Malaysia, pp. 310-315.

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

[10] A. Mesbah, A. van Deursen, and D. Roest, "Invariant-Based Automatic Testing of Modern Web Applications", IEEE Trans. on Software Eng., Vol. 38, No. 1 (Jan-Feb 2012), pp. 35-53, IEEE Computer Society.

[11] D. Amalfitano, A. R. Fasolino, A. Polcaro, and P. Tramontana, "The DynaRIA tool for the comprehension of Ajax web applications by dynamic analysis", Innovations in Systems and Software Eng., Apr 2013, Springer-Verlag.

[12] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. Carmine, and G. Imparato, "A Toolset for GUI Testing of Android Applications", 28th IEEE Int. Conf. on Software Maintenance (ICSM), 23-28 Sep 2012, Trento, Italy, pp. 650-653.

[13] B. Nguyen, B. Robbins, I. Banerjee, and A.M. Memon, "GUITAR: an innovative tool for automated testing of GUI-driven software", Automated Software Eng., Vol. 21, No. 1 (Mar 2013), pp. 65-105, Springer US.

[14] A.M. Memon and Q. Xie, "Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software", IEEE Trans. Software Eng., Vol. 31, No. 10 (Oct 2005), pp. 884-896, IEEE Press, NJ, USA.

[15] Q. Xie and A. M. Memon, "Rapid crash testing for continuously evolving GUI-based software applications", Proc. 21st IEEE Int. Conf. on Software Maintenance (ICSM'05), 25-30 Sep 2005, Budapest, Hungary, pp. 473-482.

[16] K. Meinke and N. Walkinshaw, "Model-Based Testing and Model Inference", 5th Int. Symp. on Leveraging Applications of Formal Methods (ISOLA), 15-18 Oct 2012, Heraklion, Greece, pp. 440-443.

[17] T. Kanstrén, "Towards Trace Based Model Synthesis for Program Understanding and Test Automation", Proc. Int. Conf. on Software Eng. Advances (ICSEA), 25-31 Aug 2007, Cap Esterel, France, pp. 46-55.

[18] Jemmy, an open source Java library for GUI automation, http://jemmy.java.net

[19] Microsoft UI Automation, http://msdn.microsoft.com/en-us/library/ms747327(v=vs.110).aspx

[20] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes", ACM Trans. on the Web (TWEB), Vol. 6, No. 1 (Mar 2012), article no. 3, ACM New York, NY, USA.

[21] A.M. Memon, I. Banerjee, B. Nguyen, and B. Robbins, "The First Decade of GUI Ripping: Extensions, Applications, and Broader Impacts", Proc. 20th Working Conf. on Reverse Engineering (WCRE), 14-17 Oct 2013, Koblenz, Germany, pp. 11-20.

[22] P. Aho, N. Menz, T. Räty, and I. Schieferdecker, "Automated Java GUI modeling for model-based testing purposes", 8th Int. Conf. on Information Technology : New Generations (ITNG), 11-13 Apr 2011, Las Vegas, Nevada, USA, pp. 268-273.

[23] F. Gross, G. Fraser, and A. Zeller, "EXSYST: Search-Based GUI Testing", 2012 34th Int. Conf. on Software Engineering (ICSE 2012), 2-9 Jun 2012, Zurich, Switzerland, pp. 1423-1426.

[24] W. Yang, M.R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications", Proc. 16th Int. Conf. on Fundamental Approaches to Software Engineering (FASE'13), 16-24 Mar 2013, Rome, Italy, pp. 250-265.

[25] T. Azim and I. Neamtiu, "Targeted and Depth-first Exploration for Systematic Testing of Android Apps", 2013 ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), 26-31 Oct 2013, Indianapolis, USA, pp. 641-660.

[26] A. Kull, "Automatic GUI Model Generation: State of the Art", Proc. 2012 IEEE 23rd Int. Symposium on Software Reliability Engineering Workshops (ISSREW), 27-30 Nov 2012, Dallas, TX, USA, pp. 207-212.

[27] P.Aho, T. Kanstrén, T. Räty, and J. Röning, "Automated Extraction of GUI Models for Testing", Advances in Computers, Vol. 95, Elsevier Inc, 2014, pp. 49-112.

[28] A. M. Memon, "GUI Testing: Pitfalls and Process", Computer, Vol. 35, No. 8 (Aug 2002), pp. 87-88, IEEE Computer Society.

[29] J. Strecker and A.M. Memon, "Accounting for Defect Characteristics in Evaluations of Testing Techniques", ACM Trans. on Software Engineering and Methodology (TOSEM), Vol. 21, No. 3 (Jun 2012), article no. 17, ACM New York, NY, USA.

[30] X. Yuan, M. Cohen, and A.M. Memon, "GUI Interaction Testing: Incorporating Event Context", IEEE Trans. on Software Engineering, Vol. 37, No. 4 (Jul-Aug 2011), pp. 559-574, IEEE Computer Society.

[31] A.M. Memon and Q. Xie, "Using Transient/Persistent Errors to Develop Automated Test Oracles for Event-Driven Software", Proc. 19th IEEE Int. Conf. on Automated Software Engineering (ASE), 20-24 Sep 2004, Linz, Austria, pp. 186-195.

[32] A. M. Memon, I. Banerjee, and A. Nagarajan, "What Test Oracle Should I Use for Effective GUI Testing?", 18th IEEE Int. Conf. on Automated Software Eng. (ASE), 6-10 Oct 2003, Montreal, Canada, pp. 164-173.

[33] X. Yang, "Graphic User Interface Modelling and Testing Automation", PhD thesis, Victoria University, Melbourne, Australia, May 2011.

[34] P. Aho, N. Menz, and T. Räty, "Dynamic Reverse Engineering of GUI Models for Testing", Proc. 2013 Int. Conf. on Control, Decision and Information Technologies (CoDIT'13), 6-8 May 2013, Hammamet, Tunisia, pp. 441-447.