Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing

Pekka Aho
VTT Technical Research Centre of Finland
Oulu, Finland

Matias Suarez
F-Secure Ltd
Helsinki, Finland

Teemu Kanstrén
VTT, Oulu, Finland
University of Toronto
Toronto, Canada

Atif M. Memon
University of Maryland
College Park, MD, USA

Abstract— **One of the main challenges in adopting model-based testing (MBT) is the effort and expertise required to produce the formal models. For an existing system, there are various approaches to automate the process of creating the models. In this paper, we share our experiences from a long term industrial evaluation on automatically extracting models of graphical user interface (GUI) applications and utilizing the extracted models to automate and support GUI testing. While model extraction and GUI testing has been recently a popular research topic, most proposed approaches have limitations on what can be modeled and industry adoption has been lacking. We describe the process of using Murphy tools to extract GUI models and utilize these models to automate and support various testing activities. During the evaluation, test engineers of an industrial software company used Murphy tools to support their daily efforts in testing commercial software products during 1 year time period. The results from the evaluation were promising, significantly reducing time and effort required for GUI testing.**

Keywords—graphical user interface; GUI test automation; model extraction; reverse engineering; industrial test environment;

I. Introduction

Increasing and ubiquitous use of software systems makes our daily lives more and more dependent on the software functioning without errors. The software systems are constantly growing in size and complexity and there is pressure to deliver the systems in shorter time, increasing the possibility of errors in the systems. Software testing aims to detect the errors to allow fixing them before the products are released for the end users, but the short time-to-market reduces the already scarce resources for testing. GUIs constitute a large part of the software being developed today [1], and the size and complexity of modern GUIs [2] further increase the importance and demand for automated GUI testing.

MBT is a technique for using models as a basis for automated test generation. The industrial adoption of MBT is challenging because creating formal models requires specialized expertise and a considerable amount of effort [3], and using the models for automated testing requires mapping between the model and the actual system that was modeled [4]. When an existing system is being modeled, there are various approaches to automate the process of creating the models. Especially GUI software has been a popular domain for automatically extracting models and using the models for test automation. Unfortunately most of the existing approaches have limitations and restrictions on the GUI applications that can be modeled, and so far the industry adoption has been very limited.

In our previous work [5] we have introduced Murphy tools and our innovative platform independent technique for automatically extracting GUI models based on dynamic analysis of the GUI. In this paper we describe the process of using the Murphy tools, and share our promising results and experiences from a long term industrial evaluation showing that models of non-trivial GUI applications can be automatically extracted and utilized to automate and support GUI testing.

The presented results are based on the experiences of test engineers of F-Secure Ltd using the approach and Murphy tools in their daily work of development and testing of commercial software products. F-Secure Ltd is a software company from Finland having both client and server side products related to safety and security, such as virus protection, including applications with a GUI for the end users. Especially the client products have a wide variety of supported platforms and versions of operating systems, increasing the required effort in testing. F-Secure uses continuous integration and high level of test automation in its software development. Each internal software release has to pass a rigorous quality assurance process before reaching the phase for external release. Also the creation of virtual test environments has been automated with Dynamic Virtual Machine Provisioning Service [6] to use resources more efficiently.

As directly detectable defects, such as crashes and unhandled exceptions, can be detected without using application specific test oracles [7], Murphy begins the testing of the GUI application already during the model extraction process. Although capturing a lot of details internally, Murphy abstracts and visualizes the extracted models with screenshots of the actual GUI, and the correct behavior of the GUI application can be validated by visual inspection of the state models. In addition to regression testing, the manually validated models can be used also for conformance testing, and a large part of the behavior is validated already during the visual inspection.

Our experiences from a long term industrial evaluation show that model extraction can be successfully utilized on non-trivial GUI applications and the extracted models can be utilized to automate and support various testing activities. Using the extracted models to derive GUI test scripts to replace equal manually written scripts significantly reduced the amount of hand written code related to test automation, reducing the effort required for creating and updating the test cases. Additionally, utilizing the extracted models and Murphy tools to support manual GUI testing reduced the time required for executing the existing test cases by automating the initialization phase of the test cases.

II. Background and Related Work

A. GUI Test Automation

The use of capture/replay (CR) tools has been a popular approach to automate GUI testing in industry. While CR tools are an easy and straightforward first step towards more effective use of the testing recourses [8], a significant effort is still required to record the test cases, it is challenging to detect the failures [1], and a large number of test cases may have to be re-recorded when the layout of GUI changes [9]. With the iterative development processes and rapid prototyping cycles of GUI development, the GUI changes very often, increasing the maintenance effort of CR test sets.

Model-based testing aims to reduce the test maintenance effort due to fewer artifacts to update. MBT is a technique for deriving test cases from models of the system under test (SUT) to provide more cost-effective means for extensive testing of complex systems [10]. Instead of manually writing a large set of test cases, a smaller set of test models are constructed to describe the behavior of the SUT and how it should be tested. Test cases are then automatically derived from the models, according to the selected criteria [11]. If the model describes the correct behavior of the SUT, it can be used for generating test oracles and detecting incorrect behavior by looking for situations which violate the model [9]. One barrier in industrial adoption of MBT is the complexity of modeling, requiring deep expertise in formal methods and a considerable amount of effort [3]. Another challenge is to provide mapping between the model and the implementation to be able to automatically execute the generated test cases [4].

When an existing system is being modeled, there are various approaches aiming to reduce the effort and expertise required for constructing the test models by automating some parts of the modeling process, e.g., extracting the models through reverse engineering or specification mining. Especially in the area of GUI software, there are promising academic approaches to automatically construct GUI models based on observing the runtime behavior of an existing application and using the models to automate GUI testing.

B. Automated Extraction of GUI Models

It is difficult to deduce the behavior of a GUI application from its source code without executing it, because the widgets are often reachable only from a particular state or with other constraints. The relation between GUI controls and the corresponding event handlers, and even the structure of the GUI, might be defined dynamically at runtime, with only a basic skeleton of the GUI defined statically in the source code [12]. Dynamic analysis that involves executing the application and observing the runtime behavior of the GUI is better suited for extracting models but more difficult to automate [4]. Automated execution of a GUI application requires GUI automation, i.e., simulating an end-user by automated control and interaction through the GUI of the executed application [8]. Dynamic analysis allows modeling the behavior of dynamically changing GUIs, e.g., when the visibility of a GUI component depends on the state of another component [8].

A major challenge in automatically traversing or crawling through the GUI during dynamic analysis is providing application specific input for the input fields of the GUI without predefined instructions from the user [13]. Usually, some human intervention, such as providing a valid username and password for a login screen, is required during the modeling process to reach all parts of the GUI and achieve a good coverage with dynamically extracted models [13]. Another option is that an expert manually reviews, corrects, or extends the extracted models [14]. The efficiency of these semi-automatic modeling techniques depends largely on the degree of required human intervention [14].

Memon et al. have extensively published their research on GUI Ripping [15], a technique for dynamically extracts event based models of GUI applications for test automation purposes. Their aim is to provide tools for fully automated model extraction and test generation process, so they don't address the challenge of providing specific input. Miao and Yang propose a finite-state machine (FSM) based GUI Test Automation Model (GuiTam) [16] and tooling for automatically constructing the state models by dynamic analysis. Mesbah et al. [17] present a technique and open source tool called Crawljax for crawling Asynchronous JavaScript and XML (AJAX) based applications to dynamically infer state based models. Aho et al. present GUI Driver [18], a dynamic reverse engineering tool for Java GUI applications, and an iterative process of manually providing valid input values and automatically improving the created models [13]. Morgado et al. present a fully automated version of ReGUI tool [19] and use dynamic analysis to generate GUI models in various formats. Amalfitano et al. provide tools for automated modeling and testing of rich internet applications (RIAs) [20] and Android applications [21]. Mariani et al. present AutoBlackTest [22], a tool using dynamic analysis for model extraction and test suite generation for GUI applications.

The latest research has utilized hybrid techniques, combining static and dynamic analysis. Dynamic analysis alone might miss relevant aspects of the user interface and be ambiguous regarding what conditions trigger which alternative behaviors. Then static analysis can be used to complement the dynamic analysis [12]. Yang et al. [23] proposed a hybrid reverse engineering approach and a tool called Orbit to extract GUI models of Android mobile applications. Static analysis of the application's source code is used to extract the set of user actions supported by each widget in the GUI, and the extracted actions are used to dynamically crawl through the GUI and dynamically reverse engineer a FSM model of the application.

Azim and Neamtiu [24] present Automatic Android App Explorer (A3E), a hybrid reverse engineering approach and open-source tool for systematically exploring Android mobile applications. Gross et al. [25] presented EXSYST, a hybrid model extraction and test generation tool that uses dynamic analysis for exploring Java GUI applications while guided by static analysis aiming to maximize the code coverage of the generated test suite. Silva and Campos [12] combine dynamic analysis with static source code analysis for reverse engineering Web applications.

Unfortunately all of these approaches have limitations on the GUI applications that can be modeled, most being able to extract the model of an application only if it has been implemented with a specific programming language, such as Java. In this paper, we use Murphy tools utilizing dynamic analysis and a combination of techniques to extract the GUI models, including a platform independent technique based on automatically captured screenshot images [5].

C. *Utilizing Generated Models for GUI Testing*

A challenge in automated GUI testing, especially when using extracted models, is to provide meaningful test oracle information to determine whether a test case passed or failed [26]. The input of a GUI test case may consist of a long sequence of actions, and there is no single specific output as each executed action may affect the state of the GUI. In model-based GUI testing (MBGT), the oracle information consists of a set of observed properties of all the windows and widgets of the GUI [27]. Also, to detect errors in the middle of a test sequence, the correct state of the GUI has to be verified after each executed action [26]. An incorrect GUI state during a test sequence can lead to an unexpected screen, making further test case execution useless or impossible [26].

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, 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, i.e., validation and verification against the specifications, is problematic [14]. Some defects, such as crashes and unhandled exceptions, can be detected without the use of application specific test oracles [7], making it possible to begin the testing of the GUI application already during the dynamic reverse engineering process, as in [13].

There are various types of models used for model-based GUI testing (MBGT), the most popular being state based models [28]. 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 [28]. The abstract states of a state machine are used to verify the concrete states of the corresponding GUI application during the test case execution [2]. Reverse engineered state based models are used for testing GUI applications in various approaches, e.g., GUI Driver [13] and GuiTam [16] for Java GUI applications, Crawljax [29] and DynaRIA [20] for rich internet applications (RIAs), and AndroidRipper [21] for Android applications.

Also the Murphy tools use state based models, but in addition to GUI automation frameworks, such as Jemmy Java library and Microsoft UI Automation framework, the automatically captured screenshots are utilized in distinguishing the states of the GUI. The application specific input, such as usernames and passwords, can be provided using the model extraction script before modeling or the Web UI of Murphy tools during testing.

Another popular format for extracted GUI models for testing is event based models. Memon's group has implemented GUITAR [30], 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. They also present DART [31], 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. [32] 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.

Although validated by modeling and testing open source applications and simple proof of concepts, so far none of the academic approaches and tools has been adopted by the industry to test commercial software products. An important contribution of this paper is sharing experiences from a long term industrial evaluation, and showing that model extraction can be successfully utilized on non-trivial GUI applications and the extracted models can be utilized to automate and support various GUI testing activities.

III. Utilizing Extracted GUI Models in Industrial Software Testing

In this section we introduce a process of using Murphy tools to extract models of GUI applications and utilizing the extracted models to support various GUI testing activities, and share our experiences of using the approach in industrial development and testing environment. The experiences are based on evaluation of 3 test engineers of F-Secure Ltd using Murphy tools for modeling and testing several versions of 3 commercial GUI applications during one year time period. 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).

A. *The Process of Using Murphy Tools*

At a high level, the process of using Murphy tools can be divided into two phases: 1) automated extraction of GUI models, and 2) utilizing the models in GUI testing. In our previous work [5] we have covered a large part of the first phase, so we explain the model extraction steps with less detail.

Step 1: The model extraction phase begins with writing an application specific model extraction script, usually between 2 and 200 lines of Python code, to instruct Murphy tool how to extract the model and what are the boundaries of the GUI exploration, for example ignoring the Web browser that is launched from 'Help' menu. An example of a simple extraction script is presented in Figure 1. This simple example of a GUI application, installation of 7zip application, required only two lines of Python code: one for selecting the application to be extracted and the other for setting a boundary for the extraction to the dialog for selecting the installation folder.

```
extractor = base_extractor.BaseExtractor('7zipWinScraper', '7z920.exe')
extractor.add_boundary_node('Browse For Folder')
```

Fig. 1. An extraction script of Murphy for a simple GUI application.

Step 2: The second step is letting Murphy tool to explore the GUI application and dynamically extract the model. No manual assistance or guidance is required during the model extraction. The duration of this step depends on the size and complexity of the GUI being modeled, but with the non-trivial commercial GUI applications used in this evaluation, it took about 1-2 hours to crawl through and extract the GUI model on a normal desktop PC. A simple example, the installation flow of 7zip extracted with Murphy, is presented in Figure 2. The model was extracted with the script presented in Figure 1. The transitions from a boundary node, illustrated with question marks, are included in the extracted model but not executed during the model extraction.

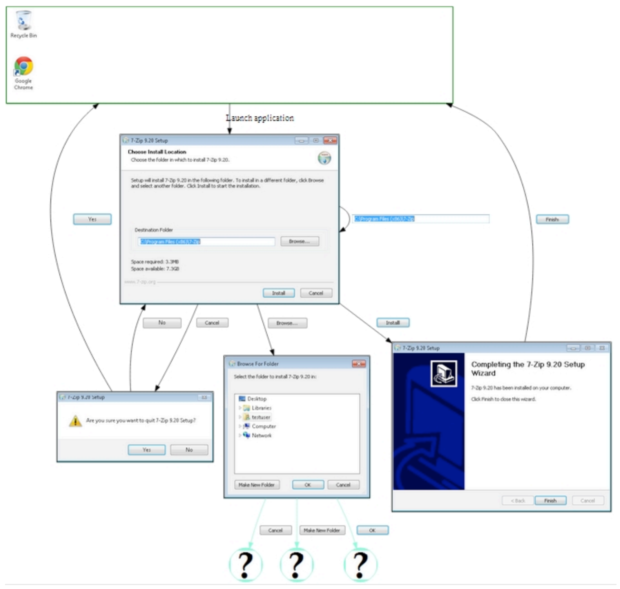


Fig. 2. A simple example of a GUI model extracted by Murphy.

Step 3: The third step of the model extraction phase is to visually inspect the model and validate the correctness of the observed behavior and extracted model. Because the model is based on the observed implementation, instead of requirements of the system, visual inspection and manual approval of the model is required to make sure that the modeled application behaves as expected. To help reading the models and understanding the behavior of the modeled application during the manual inspection, Murphy abstracts a lot of details captured in the internal model and visualizes the GUI model as a directed graph with screenshots of each GUI state as nodes and images of executed widgets as transitions between the states. The user inspecting the model should compare the modeled behavior with the correct behavior captured in the requirements or other specifications.

Usually, a couple of iterations of steps 1-3 are required to tweak and fix the extraction script and extract model with a good coverage of the GUI. After the automated model extraction, about 6 hours of manual effort was required to get an approved model.

A large part of the behavior of the GUI is already tested and directly detectable defects, such as crashes and unhandled exceptions, are found during the model extraction phase. Incorrect behavior, usually found with conformance testing, can be detected when the extracted model is manually inspected and validated. When the extracted model is inspected and approved, it can be used for automating and supporting various testing activities.

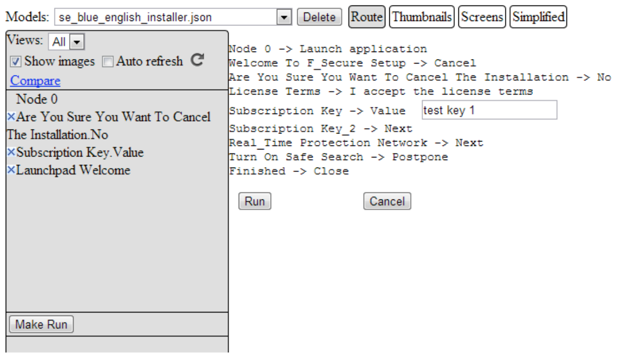
In phase 2, the extracted models can be utilized for the following three types of testing activities in any order, independently of each other.

Regression testing: The extracted models can be used for regression testing between different versions of the same GUI application. In a continuous integration process, the same extraction script can be used to automatically extract a model of the latest development version several times a day, and automatically compare the models and send warnings if changes in the behavior are detected. With major releases, the model extraction scripts might require minor modifications, such as pointing the script to the correct version of the application and to use the correct and valid product keys. When changes are detected, Murphy provides a Web UI for the user, showing the screenshots of both versions for each state having differences, highlighting the changes, and asking the user if the changes were desired new behavior or undesired deviations and faulty behavior.

Generating test scripts: When introducing new test automation tools into use, the first step is often to replace the existing, possibly manually constructed but automatically executable test cases. Murphy provides a Web UI for the user to specify test cases as paths in the extracted model, and generates executable test scripts that cover the selected path. Murphy visualizes the model with screenshots of the GUI, allows user to select states of the model and provide specific input for the test case, and randomly generates the missing parts of the path, if any. The generated test scripts can be used for example in smoke testing, automatically executed after each code commit. The amount of manually written test related code, and therefore the maintenance effort, is reduced and creating new test scripts is faster and easier.

Supporting manual GUI testing: It is seldom feasible to automate all GUI testing, and therefore Murphy provides support also for manual GUI testing. The user can use Murphy Web UI to select a GUI state from the visualized model, and

Murphy automatically creates a virtual machine, starts and executes the GUI application to the selected state, and directs the user to connect to the virtual machine with the GUI application in the desired state. The user can select a path of multiple states that should be visited and possibly the input values to be entered into the input fields of the GUI application. Murphy executes the application to visit all the selected states, automatically deducing the route if the whole path between the states was not defined. The user can continue to use the application manually, for exploratory testing or other manual verification purposes, and the time required for test initialization is reduced. An example of using Murphy Web UI to define input values for a test case is presented in Figure 3.



```
Node 0 -> Launch application
Welcome To F_Secure Setup -> Cancel
Are You Sure You Want To Cancel The Installation -> No
License Terms -> I accept the license terms
Subscription Key -> Value    test key 1
Subscription Key_2 -> Next
Real_Time Protection Network -> Next
Turn On Safe Search -> Postpone
Finished -> Close
```

Fig. 3. Defining input values for a test case with Murphy Web UI.

B. *Experiences on Extracting GUI Models*

During the evaluation, Murphy was used to extract models of 3 commercial GUI applications. Several versions of each application were modeled, because at any specific time there were at least 3 release versions on different phases of the quality assurance process, and the software developers added new features and released new versions during the development. The number of nodes (GUI states) in the extracted models was between 81 and 178.

Based on our experiences, partitioning the extracted models from one large model into a set of smaller and simpler models reduced the complexity of the models and made it easier to use the models in testing. During the experiment, separate models were created for the installation flow, the flow of actually using the application and the uninstallation flow. Also, each different supported language was modeled as a separate model. The partitioning is done with the boundaries in the model extraction scripts, and each model requires a separate extraction script with different boundaries.

Murphy was also used to extract models of all 29 different languages of the modeled products, and the visual presentations of the models were used to validate that the internationalization was working as expected. As the functionality of the application is supposed be the same regardless of the selected language, it was possible to use the same extraction script for creating the models of each language with a minor modification of changing the selected language into the extraction script. Of course, any text-based instruction in the model extraction script might be affected with the selected language and require modification.

C. *Experiences on Utilizing Extracted Models for Testing*

The extracted models and Murphy tools were used to automate and support testing and quality assurance in various ways during the software development cycle. New models of the latest versions in the continuous integration process were automatically captured and compared with earlier models 3 times a day, and warnings were sent whenever changes in the behavior were detected. Then the test engineers used the Web UI of Murphy to check each change and decide if the change was intended or an error. A large part of the detected changes were intended, and could be called false positives, but instead of having to update the related test cases, the test engineer just starts using the new model for the automated comparison. The process is similar to regression testing, but actually uses model extraction and model comparison, instead of automatically generating and executing test cases with test oracles based on behavior of the earlier version.

220 existing manually written GUI testing scripts were replaced by deriving the corresponding test cases and scripts from the extracted models. Unfortunately, precise information about the effort used for manually creating the test scripts was not available, but defining the test cases and generating the scripts from the models was obviously faster and required less effort. As a result, the amount of hand written code specific for GUI testing was significantly reduced, reducing the effort for maintaining the test cases and creating new ones in the future.

Murphy was used to support the execution of the manual GUI test cases for the modeled applications, and the time required for performing the manual GUI testing was significantly reduced. Although the reduction varied, depending on the application being tested and the particular test cases, generally the results were very promising. For example, when manual testing required over 30 minutes, it required less than 10 minutes to test the same test cases with the help of the generated models and the exploratory testing tool of Murphy. The main advantage was that Murphy automatically executed 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.

IV. DISCUSSION AND CONCLUSION

In this paper we have introduced the process of using Murphy tools and shared our promising experiences from a long term industrial evaluation on automatically extracting models of GUI applications and utilizing the extracted models to automate and support industrial testing of commercial software products. Testing of the GUI begins already during the model extraction phase, as detecting directly detectable defects, such as crashes and unhandled exceptions, does not require application specific test oracles. Incorrect behavior and errors can be detected also when the extracted model is visually inspected and validated.

Based on our experiences, utilizing extracted models to define GUI test cases and generate test scripts reduces the amount of hand written code related to test automation, reducing the effort for creating and maintaining test scripts. With the help of Murphy tools, the time required for performing manual GUI testing was significantly reduced, mainly by automating the initialization and uninteresting parts of manual testing.

Unfortunately, it was not possible to organize the evaluation in a strictly controlled and structured manner, and part of the data and metrics was confidential. As the test engineers used the approach in their daily work of development and testing of commercial software products, and their goal was to reduce the effort of GUI testing by being more efficient, they were not eager to spend time for collecting data for academic purposes. Therefore we were not able to present as accurate data and measurements of the costs and benefits of the approach as often presented from smaller scale, controlled experiments and academic feasibility studies.

We believe that an important contribution of this paper was to show that it is possible and feasible to use dynamic analysis to extract state based models of industrial size GUI applications and validate the extracted models by visual inspection, and using the extracted models for GUI testing saves time and effort compared to the existing methods and tools. A good indicator of success is also that based on the results, Murphy is being adopted into wider use in F-Secure Ltd.

REFERENCES

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

[2] A.M. Memon, "An event-flow model of GUI-based applications for testing", Software Testing, Verification & Reliability, Vol. 17, No. 3 (Sep 2007), pp. 137-157.

[3] G.Z. Holzmann and M.H. Smith, "An Automated Verification Method for Distributed Systems Software Based on Model Extraction", IEEE Trans. on Software Eng., Vol. 28, No. 4 (Apr 2002), pp. 364-377.

[4] A.M.P. Grilo, A.C.R. Paiva, and J.P. Faria, "Reverse engineering of GUI models for testing", Proc. 2010 5th Iberian Conf. on Information Systems and Technologies (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", Proc. Int. Workshop on Experiences and Empirical Studies in Software Modelling (EESSMod), 1 Oct 2013, Miami, Florida, USA, pp. 49-54.

[6] Dynamic Virtual Machine Provisioning Service, https://github.com/F-Secure/dvmps

[7] X. Yang, "Graphic User Interface Modelling and Testing Automation", PhD thesis, School of Engineering and Science, Victoria University, Melbourne, Australia, May 2011.

[8] P. Aho, N. Menz, and T. Räty, "Dynamic Reverse Engineering of GUI Models for Testing", Proc. Int. Conf. on Control, Decision and Information Tech., 6-8 May 2013, Hammamet, Tunisia, pp. 441-447.

[9] J. Bowen and S. Reeves, "UI-design driven model-based testing", Innovations in Systems and Sw.Eng., Vol. 9, No. 3 (2013), pp 201-215.

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

[11] T. Kanstrén, "A framework for observation-based modelling in model-based testing", VTT Publications 727, Espoo, Finland, 2010.

[12] C.E. Silva and J.C. Campos, "Combining static and dynamic analysis for the reverse engineering of web applications", Proc. 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS), 24-27 Jun 2013, London, UK, pp. 107-112.

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

[14] 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.

[15] 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.

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

[17] 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 (2012).

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

[19] I.Morgado, A. Paiva, and J. Faria, "Dynamic Reverse Engineering of Graphical User Interfaces", Int. Journal on Advances in Software, Vol. 5, No. 3 & 4 (2012), pp. 224-246.

[20] 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 Sw. Eng., Apr 2013.

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

[22] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro, "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications", Proc. IEEE 5th Int. Conf. on Software Testing, Verification and Validation (ICST), 17-21 Apr 2012, Montreal, Canada, pp. 81-90.

[23] 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), 16-24 Mar 2013, Rome, Italy, pp. 250-265.

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

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

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

[27] 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).

[28] 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.

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

[30] B. Nguyen, B. Robbins, I. Banerjee, and A.M. Memon, “GUITAR: an innovative tool for automated testing of GUI-driven software”, Automated Software Engineering, Vol. 21, No. 1 (2013), pp. 65-105.

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

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

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