2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops

Automatic GUI Model Generation: State of the Art

Andres Kull
Elvior LLC, Tallinn, Estonia[1]
University of Maryland, Dept. of Computer Science[2]
College Park, MD, USA[2]
andres.kull@elvior.com[1]; akull@umd.edu[2]

***Abstract* – Modeling has been one of the most significant obstacles to why model-based testing has not been taken into use by industry on a large scale. Therefore, generating models automatically is an attractive way of thinking. In well-structured application domains, such as graphical user interface (GUI), this method can be used successfully. This paper gives a state-of-the-art overview of GUI model generation methods. The paper analyzes the methods from the point of view of model-based testing of GUI applications, where generated GUI models are used for generating the tests.**

Model-based testing; automatic GUI model generation; GUI model

I. INTRODUCTION

The purpose of this paper is to provide a state-of-the-art overview of GUI model generation methods. We look at the topic from a GUI model-based testing point of view where test generator generates tests automatically from the model.

Model-based testing is a test automation approach where the scripts for automated test execution are automatically derived from the model of the system under test (SUT). Model-based testing increases the productivity in creating test scripts and improves test coverage compared to manually coded test scripts.

Model-based testing is a broad topic. We focus on model-based functional black box testing of GUI-based applications. SUT is tested automatically as a black box by controlling its input events and observing output events by the test scripts.

Model-based testing workflow consists of the following steps:

- modeling of the SUT;
- defining the required test coverage;
- generating tests automatically from the model;
- testing the system by executing generated tests against the SUT.

The two last steps in this workflow can be fully automated, and there are a lot of model-based testing methods and tools for this purpose [9].

Despite the obvious benefits of model-based testing as such, modeling itself has been one of the most significant obstacles to why model-based testing has not been taken into use by industry on a large scale. The reasons are the following:

- Modeling of the SUT has been manual. It is a labor-intensive task.
- Modeling requires analytical skills that test engineers often do not have. Introducing model-based testing in organizations requires testing teams to consist of different people from what they are used to.
- The state space of the model will increase significantly in some application domains, making model creation and maintenance too expensive compared to the implementation of the testable application itself.
- Using agile development processes, developers are iterating quickly from version to version of the implementation. Such system development starts without clearly defined requirements and the requirements themselves are evolving from iteration to iteration. Modeling for testing purposes in such an environment cannot keep pace with development.

In this paper, we focus on GUI-based applications. While the first two issues are true for all kinds of systems the last two are particularly relevant for GUI-based application development. GUI-based applications are usually created using agile processes where the requirements and GUI change a lot during the development. GUI applications are typically modeled as state machines. States represent visible states of the GUI and the transitions represent user interactions with the GUI that drive the GUI from state to state [7]. Each visible state of the GUI is defined by a set of GUI windows, widgets, parameters of the widgets, and their values. Therefore, the state space of a typical GUI tends to increase immensely, and its maintenance will be expensive compared to changes in the GUI implementation.

In order to benefit from model-based testing, automatic model generation techniques have been studied in GUI application testing with remarkable success.

This paper gives an overview of the different techniques used for generating GUI models. We outline the pros and cons of the available techniques.

II. MANUAL VS. AUTOMATIC MODELING FOR TESTING

A. Manual modeling

Modeling usually starts from the requirements that the implementation of a system has to meet. Modeling is a method of formalization of the requirements into the model that can be used for deriving the tests automatically from it. Requirements (requirement specifications, interface

978-0-7695-4928-6/12 $26.00 © 2012 IEEE
DOI 10.1109/ISSREW.2012.23

specifications, etc.) describe the expected correct behavior of the system. The model becomes the representation of the truth and the test scripts that are generated from such model check if the implementation meets its expected behavior. Such tests are called conformance tests.

Usually the requirements are given in natural language using informal textual descriptions and diagrams. Formalization of the often incomplete, inconsistent, and contradictory informal textual requirements is a task that is suitable for people. Only people with analytical skills are able to create a formal and structured model by studying different documents and discussing with other people who can help to determine the ambiguities. The program cannot accomplish this task automatically.

B. Automatic modeling

How is it possible to generate the models for testing a GUI-based system from the informal requirements automatically and to derive test scripts from this model that test the application's conformity to specifications? The answer is – it isn't. In order to generate the model automatically we have to lower our original goal. We cannot automatically generate the model that allows us to test the conformity to the implementation of the requirements. Tests generated from automatically generated models can do something else that is not as valuable but still valuable enough.

Automatic model generation is actually a reverse engineering task. Implementation of the system is the source of information for the reengineering task. One may ask which value has the test that is generated from the implementation to test the same implementation. This is a valid question. Those tests cannot automatically reveal if the system meets its requirements. Nevertheless, they can automatically test the application with much better test coverage than any person is capable of doing. They can reveal the bugs in the implementation that cause crashes, for example.

C. Semi-automatic modeling

In general fully automatic model generation is not possible or if it is used then the test coverage will remain poor. Some kind of human intervention is usually needed. By semi-automatic modeling we mean the modeling techniques that are performed mostly automatically but are either:

- assisted manually by a person during the reverse engineering process; or
- reviewed, corrected, and extended manually by a person after automatically generating the initial model.

Below we can see that most techniques require human intervention. The efficiency of the technique depends on the degree of required human intervention.

III. REVERSE ENGINEERING

As said above, automatic model generation is a reverse engineering task. Reverse engineering is extracting the design artifacts and deriving abstractions that are less implementation-dependent. In general, there are two approaches for reverse engineering. A static approach analyses the system source code or binary representation without executing the system [16].

The static approach requires access to the source code of the system, which is not always available. Static approaches are particularly well suited for extracting information about the internal structure of the system and dependencies among structural elements [16]. Reverse engineering from the source code or binary code is the most precise because all the information about the software's behavior is present in them. Reverse engineering from the source code is easier to do manually than reverse engineering from binary representation because binary code reverse engineering requires low-level programming skills. From an automatic reverse engineering point of view, source code and binary code reverse engineering complexity is similar. Both of them apply statistical analysis methods and tools on the code. The static approach has been used in reverse engineering of GUI applications by [1], [2], [3], [4], [5] and [13], for example.

A dynamic approach analyses the external behavior of the system by executing it. The dynamic approach is well suited to extract the structure of the GUI and some of its behavior, but it is more difficult to automate than static approaches [16]. The dynamic approach has been used in reverse engineering of GUI applications by [7], [12], [18], [11] and [19], for example.

IV. STATIC ANALYSIS

Static analysis of the source code is performed to figure out the important behavioral and structural elements in the source code.

GUI reverse engineering from the source code became a topic in the software engineering industry when text-based user interfaces were starting to be replaced by GUIs in the early 90s. There was a need for migrating the legacy text-based applications to GUI-based platforms. Automatic migration of the BASIC text-based user interface to the GUI platform, which includes a GUI builder and compiles event-driven C/C++ code, is described in reference [2]. They inferred a structural representation in terms of abstract graphical objects and callbacks from the source code and built a translator from BASIC to C. A knowledge-based interface migration process was described in reference [1]. In this work, the source code of the user interface of an original system was reverse engineered, the structure of the GUI was created, and a rule-based inference was applied to compile the GUI structure to target system GUI implementation. It is pointed out by [4] that GUI reverse engineering from source code for GUI migration purposes was not widely adopted because of the high risk of violating the original application internal coherence.

The emergence of mobile web introduced a need to present the same HTML content on devices with different layout sizes. Several authors used the reverse engineering approach to build a model of the existing web site that could be used for transforming the content to different platforms automatically. For example, paper [5] addressed the problem of making a web site accessible to a wide range of computing platforms. They allowed developers to translate

web pages into a model-based representation. They extract the hierarchy of the user interface elements contained in HTML pages and their layout relationships using static analysis. The authors of the paper [6] converted the visual layout of HTML forms into a semantic model with explicit captions and logical grouping. They also inferred the semantic relationships between GUI elements (such as captions and hints) automatically.

An automated static reverse engineering of GUI code for helping software maintenance and reuse is described in reference [3].

The author of reference [13] used static analysis of the source code to extract the architecture of the user interface part of an application. Apart from the GUI's architecture, the approach also extracts a graph that represents the control flow of events between the different windows of the GUI. The tool extracts the program's windows and their structure, the attributes of the widgets and their values, the GUI events that might occur at runtime, and the event handlers associated with those events. The tool is based on static program analysis techniques, such as inter-procedural pointer, control-flow, and static single assignment analysis.

The authors of reference [17] analyzed the source code and extracted the GUI-related parts to generate state machines and behavioral graphs from Java (Swing or GWT) and the Haskell application's source code. Their GUISurfer tool uses a number of techniques to achieve re-targetable reverse engineering of GUI source code. Using a parser for the relevant programming language, an Abstract Syntax Tree (AST) is obtained from source code. A number of application properties were possible to reason from the generated models. For example, graph-based reachability analysis can be used to compute if all the states are accessible from the initial one, in order to detect whether a particular window of the application will ever be displayed. Test cases that test those properties can also be generated from those models.

There are several issues that make the static analysis for reverse engineering of the GUI applications a complicated task. Understanding of the GUI structure by reverse engineering is a feasible task as shown by the referenced authors, but understanding the behavior of an object-oriented system is even more difficult than understanding its structure. Because of inheritance, polymorphism, and dynamic binding, it is difficult and sometimes even impossible to know, using only the source code, the dynamic type of an object reference and, thus, which methods are going to be executed. Multithreading and distribution further complicate the analysis [8]. Reference [17] points out that the static approach is not able to deal with continuous media and synchronization/timing constraints among objects. Also, information about overlapping windows is not available for static analysis since this must be determined at run time.

Due to the issues listed above static analysis has not been used successfully in generating GUI models for testing purposes. Dynamic analysis has been more popular and successful in reverse engineering of GUI applications.

V. Dynamic Analysis

Dynamic analysis observes the system during the execution and observes the important patterns where a model can be derived from. Dynamic analysis can be passive or active. Passive dynamic analysis just observes the inputs and outputs of a system while the active one also controls the inputs of the system.

There are several techniques how to attach the "probes" to the system under analysis for control and observation purposes. Source code instrumentation, byte code instrumentation, and GUI APIs are the most common in GUI application reverse engineering.

A. Source code instrumentation

This is a technique where the programmers implement instrumentation in the form of source code instructions so that the application can be controlled and observed by an external application. Source code instrumentation is a most powerful way to retrieve information from the application and to control it because the abstraction level of the information and the information sources are under the control of the reverse engineer. Instrumenting of the source code poses a number of problems that significantly reduces the feasibility of it. Firstly, in most cases access to the source code is not available. Secondly, if the source code is available then the developers face a dilemma: to keep only the clean (original) version or the instrumented version or keep both versions of the source code. Both options have disadvantages.

B. Byte code instrumentation

To avoid repeated instrumentation, cleaning of the source code and possible inconsistencies between the instrumented and non-instrumented versions of the implementation, the byte code instrumentation can be used. Byte code instrumentation is less intrusive than source code instrumentation. In Java applications reverse engineering the byte code instrumentation can be used by applying aspect oriented programming and AspectJ as demonstrated by [8] and [14].

C. Using GUI APIs

Platform and GUI-specific APIs can be used to observe GUI state information and control the traversal of the GUI from state to state.

In CelLEST project [4] interaction-based reengineering for migration of GUI applications was used. The GUI of the original application was wrapped with a software layer that interacted with the web interface and hid the original one. The wrapper used the API provided by the original application. For building the wrapper, a source code-independent method was used to model the interaction between the original user interface and its user. The traces were recorded, the user interface snapshots where taken, and the state machine was built from the states. The model was mostly created for understanding the original GUI behavior in order to build the correct wrapper.

Java GUI APIs are used to retrieve GUI state information during the execution of Java GUI applications and to control

the execution of the application for model exploration purposes by [7], [18], and [19].

VI. GUI Model Exploration Techniques

For exploring the model of the implementation the implementation inputs have to be stimulated and the outputs have to be observed. Based on the explored information the model is built up piece-by-piece. Many techniques are available for model exploration. The main idea in GUI model exploration is automatically to traverse through the GUI, to explore every important GUI state and to store information about the GUI structure (structural model) and flow of events connected to the state information (state model).

A. Manual model exploration

The easiest model exploration technique for GUI-based applications is manual exploration by the user, who walks through the application while the background process is recording user actions and GUI state changes automatically. All capture/playback GUI test tools use such manual walkthroughs. It is well known that a manual walkthrough is time-consuming, and its exploring ability is poor.

Nevertheless, in reference [12] the manual walkthrough to generate probabilistic finite state model representation of a GUI application that combines its static structure with actual usage data is described. The probabilistic model was used for generating test cases that were able to find more errors than the initial walkthrough.

B. Fully automated model exploration

The authors of reference [7] proposed a fully automated dynamic GUI model exploration technique for generating GUI models for test generation purposes. GUI ripping obtains models of the GUI's structure and execution behavior automatically. The process of GUI ripping works in the following way. Starting from the first window of a GUI, the GUI is explored by opening all child windows. All widgets of the windows, properties of the widgets and their values are extracted. GUI ripping extracts both the structure and execution behavior of the GUI. The GUI's structure is represented as a GUI forest and its execution behavior as an event-flow graph and an integration tree. Each node of the GUI forest represents a window and encapsulates all the widgets, properties and values in that window. Event-flow graphs (EFG) and the integration tree show the flow of events in the GUI. The execution model of the GUI is obtained by using a classification of the GUI's events. Once a test designer verifies the extracted information, it is used automatically to generate test cases.

The fully automated process described in [7] has some weaknesses. It came out that not all the GUI behavior was possible to traverse due to the infeasible paths. The presence of infeasible paths in GUIs prevents the full automation. For example, some windows may be available only after a valid password has been provided. Since the GUI ripper may not have access to the password, it may not be able to extract information behind the authentication window. It is also possible that some panels (or modeless windows) are sometimes visible and sometimes invisible. Toggling the visibility-property value of a panel can cause some events to be exposed or hidden (controls in the newly enabled panel or modeless window).

Therefore, it means that the fully automated model is far from being complete. Major parts of the application may be missing from the explored model. To overcome the issue [7] provided additional tool support to add manually missing parts to the fully automatically explored GUI model.

Miao and Yang [18] challenged the work of Memon [7] by proposing the GUI model and its generation algorithm that is in some extent more efficient than Memon's model. They proposed an FSM-based GUI Test Automation Model (FSM based GuiTam), which can automate all the tests of the EFG-based approach. Instead of different structural and state models they generated one state model where the nodes represent GUI states and the arcs represent the events from one state to another. They proved that the efficiency of the new model, in terms of storage and computational complexity, is at least as good as that of EFG GUI test automation. Furthermore, this model is also able to automate GUI tests in the scenarios of a "non-fixed events set" and "expandable panel", which cannot be addressed by the EFG model. Reference [18] proved that, on average, the space complexity of the GuiTam model is of O(n), which is one order less than that of the given EFG. The computational complexity of generating test cases for both of the two models is of the same order.

Reference [15] uses dynamical analysis to generate a model for generating tests for AJAX user interfaces. Their method is based on the crawler to infer a flow-graph for all (client-side) user interface states. They added to the model automatically AJAX domain-specific information for also generating a test oracle automatically. Therefore, they identify AJAX-specific faults that can occur in such states (related to DOM validity, error messages, discoverability, back-button compatibility, etc.) as well as DOM-tree invariants that can serve as an oracle to detect such faults. Mesbach's work was successful mainly because of the well-structured underlying domain artifacts. CRAWLJAX can exercise client side code, and identify clickable elements that change the state within the browser's dynamically built DOM. From these state changes, they infer a state-flow graph, which captures the states of the user interface, and the possible event-based transitions between the states.

As a summary, the most challenging issues that the fully automated model exploration faces are the following. First, it is impossible to generate automatically user inputs for text fields that will allow exploring more GUI state space just by using the dynamic model analysis. Second, it is hard to find sometimes automatically the ways to open all windows. For example, some menu items should be first enabled for that. There are two ways to overcome the issues – either to perform some additional static analysis of the code to find the constraints on the infeasible paths and solve them automatically or to use user-assisted automatic model exploration. The author of this paper is not aware the applications of the former one. The latter one is described in the next subchapter.

C. User-assisted automated model exploration

Reference [16] proposed an algorithm of the exploration that mixes automatic exploration with user-assisted manual exploration. The algorithm has three phases: the first one extracts structural information about each window of the GUI under test and asks the tester to open/close all windows while saving his steps to open/close windows in the next phases of the process. The tool does not interact with the GUI. It only reads information about the structure (GUI controls) of the windows opened by the tester. The second and third phases extract behavior information by interacting with the GUI automatically without the help of the tester.

The authors of reference [16] describe another approach of mixing the manual and automatic exploration. They generate automatically by the exploration process a skeleton of a state machine model of the GUI, represented in a formal pre/post specification language. Mapping information between the model and the implementation is also generated along the way. The automatically extracted model is then completed manually in order to get an executable model. Infeasible paths are solved in the model manually. Also, the information for generating the test oracle is added manually to the model. The model is used to generate abstract test cases, including the expected outputs, automatically from the final model and executed over the GUI application, using the mapping information generated during the exploration process.

Reference [19] uses the Java GUI ripping technique that is similar to the technique in [7] for retrieving runtime information from the Java GUI application and for controlling the automatic exploration. For GUI model representation, they use the model proposed by [18]. Reference [19] targets specially the issues of model exploration by providing an efficient tool for the user who assists the automatic process of model exploration. The models are generated while automatically executing and observing the SUT. The generated models include structural tree models presenting the GUI components and their properties, and a GUI state model presenting the behavior of the GUI and mapping the structural models into the abstract states. Structural models are generated for each state of the GUI application. Their tool provides a user-friendly way to reduce the effort required to create the models, supports human readable graphical models that can be refined manually by user, for example, inserting valid input values for specific input fields of the GUI application. Also, the created graphical models of the SUT allow checking the implementation against the requirements of the system. Comparing two consecutive structural models makes it possible to observe the changes happening in the GUI while automatically interacting with the GUI application. The strengths of their approach compared to the earlier work include automatically generating human readable graphical models while requiring none or only a little manual effort to complete the models.

Aho et al. go further in their next paper [20] by providing better tools for users in assisting the automatic model exploration. Combining valid test data with the generated model was not well addressed in earlier approaches. An approach, presented in [20] provides a practical way to combine manually given test data into the automated GUI model exploration process. Their GUI Driver tool is extended to support the iterative process of automated modeling and manually augmenting the model with valid test data. Each round of the iterative process includes an automated modeling phase and a manual phase of providing valid test data. The GUI Driver tool allows the user to provide the valid input in a practical and easy way, using the input fields of the GUI application in the same way as an end user of the application would. Because of this approach the GUI Driver tool is able to reach and model more states of the GUI applications resulting in more accurate GUI models. The tool supports re-using and updating the test data when the GUI application changes.

VII. GUI Model Refinement Using Feedback

Models that are generated using either static or dynamic reverse engineering techniques and tests generated from those models are as good as the applied technique is able to generate. Those models and respectively the generated tests can be improved by increasing the ability to cover the application more exhaustively, faster and with fewer computational resources. Several feedback-based techniques are proposed for these purposes.

These techniques require an initial test case/suite to be created, either manually or automatically, and to be executed on the software. Feedback from the execution is used automatically to refine the model that allows reaching better test coverage with smaller test generation and execution costs [10]. Different sources of feedback are possible for the model refinement. An example of feedback in unit testing (applicable in other test phases as well) is a code coverage report. The report pinpoints the branches in the source code that were not covered with the tests and that need additional tests to be created.

Reference [10] describes an approach that analyses the trace information collected during the test execution and refines the model that allows additional test cases to be derived that were not possible from the original model. The approach relies on a hypothesis that events that are handled by the interacting event handlers should be tested together. Interacting event handlers are the ones that share variables, exchange messages, share data, etc. A seed test suite is generated automatically using an existing event-interaction graph (EIG) model of the GUI. The seed suite is executed on the GUI automatically. During test execution, the runtime state of GUI widgets is collected and used automatically to identify an Event Semantic Interaction (ESI) relationship between pairs of events. This ESI relationship actually identifies complex interactions among GUI event handlers and gives information on which events should be tested together by the same test case because the same GUI handlers most probably handle them. Further a new model of the GUI, called the Event Semantic Interaction Graph (ESIG), is constructed automatically. It is used to define the test coverage for generating additional test cases that take

into account the ESI relationship. This feedback loop can be repeated incrementally.

VIII. TEST ORACLES

Test models created using reverse engineering of any kind in general lack the test information for generating test oracles. The information for test oracle is not available just by observing the execution of the implementation using dynamic reverse engineering. The information gained by static analysis from implementation source code cannot be used for validating the implementation itself.

In order to improve the model so that the test oracle could be generated automatically, the model has to be refined by adding some more information.

The authors of [15] were able to generate oracles from the model automatically. It was feasible because the domain specific know-how for test oracles was possible to add automatically into the model during the model creation. This was the additional information that was hard-coded into the test generator. Such approach is not possible for a general-purpose model generator that is independent of the application domain.

If the model is generated using some reverse engineering technique then only a person can refine the model manually by adding some additional information to the initial model. The authors of [11] modified the SpecExplorer model manually for this purpose. The authors of [7] and [18] used a "golden log" from the correct, verified implementation execution as a test oracle for subsequent regression tests.

IX. CONCLUSIONS

Static and dynamic approaches for the reverse engineering of GUI applications both have a number of advantages and drawbacks. The static approach is able to retrieve more accurate and complete information from the implementation about what could happen in runtime. The dynamic object-oriented nature of the GUI application, on the other hand, complicates the analysis. The dynamic approach has been easier to implement for GUI applications, but, on the other hand, the accuracy of a dynamic approach depends on the inputs able to be used in controlling the model exploration.

State-of-the-art shows that dynamic reverse engineering techniques have been able to show better results for automatically generating models for GUI testing as the techniques rely on static analysis.

The most challenging issue in dynamic reverse engineering is caused by an inability to explore the GUI model fully automatically. Solving infeasible paths and providing user inputs have been the tasks that still require human intervention. The current research in dynamic reverse engineering is trying to provide more efficient and user-friendly tools to reduce the time that users should spend in assisting the automatic model-exploration [20].

REFERENCES

[1] M. Moore, S. Rugaber, and P. Seaver, "Knowledge-based user interface migration," Proc. of the 1994 International Conference on Software Maintenance, pp. 72-79, 1994.

[2] G. Antoniol, R. Fiutem, E. Merlo, P. Tonella, "Application and user interface migration from BASIC to Visual C++," Proc. 11th International Conference on Software Maintenance (ICSM'95), 1995.

[3] A. Michail, "Browsing and searching source code of applications written using a GUI framework," Proc. of the 24th International Conference on Software Engineering (ICSE), pp. 327-337, 2002.

[4] E. Stroulia, M. El-Ramly, and P. Sorenson, "From legacy to web through interaction modeling," Proc. 18th IEEE International Conference on Software Maintenance, ICSM, 2002.

[5] L. Bouillon, J. Vonderdonckt, and N. Souchon, "Recovering alternative presentation models of a web page with VAQUITA," Proc. CADUI '02, pp. 311-322, 2002.

[6] Y. Gaeremynck, L. Bergman, and T. Lau, "MORE for less: model recovery from visual interfaces for multi-device application design," Proc. 8th International conference on Intelligent user interfaces, January 12-15, 2003, Miami, Florida, USA, 2003.

[7] A. M. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: reverse engineering of graphical user interfaces for testing," Proc. 10th Working Conference on Reverse Engineering, Nov. 2003.

[8] L. Briand, Y. Labiche, and J. Leduc, "Towards the reverse engineering of UML Sequence Diagrams for distributed Java software," IEEE Transactions on Software Engineering, Vol. 32, no. 9, pp. 642-663, Sept., 2006.

[9] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, ISBN-13:9780123725011. Elsevier Science & Technology Books, 2006.

[10] X. Yuan, A. M. Memon, "Using GUI run-time state as feedback to generate test cases," Proc. 29th International Conference on Software Engineering, (Washington, DC, USA), May 23-25, 2007, pp. 396-405, 2007

[11] A. C. R. Paiva, J.C.P. Faria, and P. Mendes, "Reverse engineering formal models for GUI testing," Proc. 12th International Workshop on Formal Methods for Industrial Critical Systems (FMICS), Berlin, Germany, 2007.

[12] P. Brooks, A. M. Memon, "Automated GUI testing guided by usage profiles," Proc. 22nd IEEE International Conference on Automated Software Engineering, (Washington, DC, USA), 2007.

[13] S. Staiger, "Reverse engineering of graphical user interfaces using static analyses," Proc. 14th Working Conference on Reverse Engineering, Vancouver, BC, Canada, 29-31 October 2007.

[14] H. Samir, A. Kamel, "Automated reverse engineering of Java graphical user interfaces for web migration," Proc. ITI 5th International Conference on Information and Communications Technology (ICICT), Cairo, Egypt, December 2007.

[15] A. Mesbah, A. van Deursen, "Invariant-based testing of Ajax user interfaces," Proc. Int'l. Conf. on Software Eng. (ICSE), 2009.

[16] A. M. P. Grilo, "Reverse engineering of GUI models for testing," Proc. 5th Iberian Conference on Information Systems and Technologies (CISTI), 2010

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

[18] Y. Miao, X. Yang, "An FSM-based GUI test automation model," Proc. 11th International Conference on Control, Automation, Robotics and Vision (ICARCV 2010), Singapore, 7-10th December 2010.

[19] P. Aho, N. Menz, T. Räty, and I. Schieferdecker, "Automated Java GUI modeling for model-based testing purposes," Proc. 8th International Conference on Information Technology: New Generations (ITNG 2011), April 11-13, 2011, Las Vegas, Nevada, USA, 2011.

[20] P. Aho, N. Menz, and T. Räty, "Enhancing generated Java GUI models with valid test data," Proc. IEEE Conference on Open Systems (ICOS), 2011.