TOM: a Model-Based GUI Testing framework

Miguel Pinto, Marcelo Gonçalves, Paolo Masci, and José Creissac Campos

HASLab/INESC TEC & Dep. Informática/Universidade do Minho, Braga, Portugal

Abstract. Applying model-based testing to interactive systems enables the systematic testing of the system by automatically simulating user actions on the user interface. It reduces the cost of (expensive) user testing by identifying implementations errors without the involvement of human users, but raises a number of specific challenges, such as how to achieve good coverage of the actual use of the system during the testing process. This paper describes TOM, a model-based testing framework that uses a combination of tools and mutation testing techniques to maximize testing of user interface behaviors.

Keywords: Model-based testing; User Interfaces; Tool support.

1 Introduction

User interface testing is an important aspect of interactive computing systems development, and a range of techniques can be useful in this context. Analysis techniques based on user experiments are mostly concerned with assessing the quality from the users' perspective (e.g. satisfaction, reliability, learnability, efficiency – cf. the notion of usability [7]). They can be used to explore a limited number of scenarios, and do not allow developers to identify all potential user interface problems. Model-based verification tools provide an alternative perspective, and enable the exhaustive analysis of core usability aspects such as mode visibility and consistency of response to user actions (cf. [4]). However, usability properties proved over the models are "inherited" by the final system implementation only if the final system faithfully implements the models.

Model-based Testing (MBT) [18] is a technology that can help bridge this gap between model-based verification and a system's implementation. It is a black-box testing technique that compares the behavior of an implemented system (called SUT, System Under Test) with that prescribed by a model of the same system (the Oracle). One advantage of MBT is that it facilitates full automation of the testing process, from test case generation to test execution.

Several authors have explored a range of approaches for using MBT on user interfaces: based on reverse engineered models [6,1] or purpose built models [15] representing the UI behavior; using Oracles that capture the control dialogues implemented in the UI (e.g., to capture normative interactions between users and system [3,17]); or using predefined patterns to generate test cases based on given Oracles [12,13]. Different alternatives have also been explored for executing test cases: using code instrumentation, UI automation frameworks, or higher-level co-execution tools.

Memon was among the first to apply MBT to graphical user interfaces [11]. He developed the GUITAR [14] testing framework, which supports a variety of model-based testing techniques. The framework uses a reverse engineering process to generate

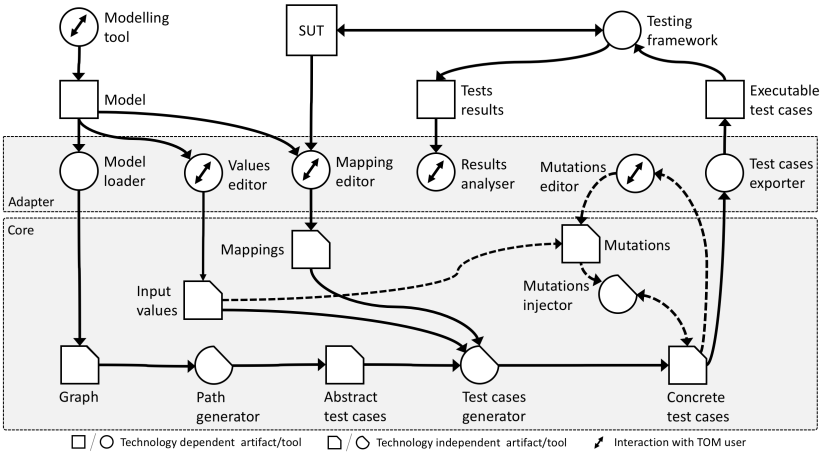

Fig. 1. TOM framework: conceptual architecture

a model of the SUT. One limitation of GUITAR and other similar tools (see [9] for a recent survey of similar tools) is that oracles focus on the SUT, making it harder to select test cases that are relevant from the user's perspective. In this work, we introduce TOM, a model-based testing framework that aims to address this gap by enabling automatic generation of user relevant test cases.

Contribution. The specific contributions of this paper are: (i) a presentation of the architecture and prototype implementation of the TOM framework; (ii) an example application of the framework to an existing Web application.

2 The TOM framework

The framework adopts a modular approach to better support the exploration of different model-based testing techniques. It is divided into two layers (see Figure 1): the Adapter Layer, and the Core Layer.

The Adapter Layer is the interface between the core of the framework, and the oracles and test automation frameworks. It includes a set of Model Loaders responsible for importing a UI model, and a set of Test Cases Exporters necessary for generating test cases. A number of other components support the user during the configuration process and while running the test cases: a Values Editor is provided to support editing of values to be used as inputs; a Mapping Editor links a state machine model of the UI to the graphical UI of the SUT; a Mutations Editor allows users to define the type of mutations to be introduced in the test cases; and a Results Analyzer presents test results.

The Core Layer uses a graph representation of the SUT to perform the generation of test cases. The generation process uses a directed graph produced by the Adapter Layer. Each node in the graph represents a dialogue unit in the interface (e.g. a modal window, or a page in a Web application), and contains information about the actual content of the dialogue unit and the validations checks to be performed over it. Edges

represent changes to the interface in response to user actions (e.g. button clicks). A Path Generator component in the Core Layer generates Abstract test cases as paths over the directed graph. These paths represent normative usage scenarios of the system. These usages are specified in a way that is independent of any specific implementation, as they are expressed over the graph. They are converted into concrete test cases by a Test Cases Generator component in the Core Layer. This component uses two additional sources of information (provided by the Adapter Layer): a mapping between the state machine and the graphical UI of the SUT, and input values for specific UI widgets. Finally, a Mutations component generates additional test cases with the aim of achieving fault coverage. The considered fault classes are based on Reason's use error types [16] (slips, lapses and mistakes). Mutations are introduced in the normative usage scenarios, either randomly or according to user defined criteria, to check the impact of these use errors.

Prototype Implementation. We have implemented an Adapter Layer for Web applications[1] as a Google Chrome extension (see Figure 2, left). The Adapter captures the user interaction with a Web page to create a first version of the model. It then supports editing the model (to add new states and transition not covered in the capture phase, define the values to be used and the validations to check), as well as the mapping between the final model and the graphical user interface. A companion component exports test cases to Selenium WebDriver, a tool to automate testing of web pages.

The directed graph representing the UI model is expressed in SCXML [19]. The main tags are: `<state>`, used to represent dialogue units; and `<transition>`, used to represent events. `<state>` tags have a type that defines the characteristics of the dialogue unit. An example state type is 'Form', which represents a modal window where a number of input fields must be input before the interaction can proceed. Validation checks are declared in the state using `<onentry>` and `<onexit>` tags, which are assessed when entering or leaving a state, respectively. Example validation checks include: `displayed?` (checks whether a given element is visible); `is_selected` (checks whether an element in a drop-down list or check-box is selected); `enabled?` (checks whether an element is enabled); `attribute` (checks the value of an attribute of an HTML element); and `regex` (checks whether an element contains a value that matches a regular expression). Transition tags can use a `<step>` attribute to decompose a logically atomic action into its constituent physical actions. Example transitions are `<select>` (the action of selecting an option in a drop-down menu); `<submit>` (the action of ending a dialogue unit) and `<error>` (events triggered in the case of errors).

The Path Generator module converts the SCXML file into a graph using the JGraphT Java library. JGraphT provides a number of traversal algorithms (from calculating the shortest path between two nodes using Dijkstra's algorithm to calculating all paths), which can be used on the graph to yield abstract test cases. Test case generation is controlled by defining a start and an end state, and upper bounds on the number of visits/traversals to nodes/edges on the graph.

The Mutation component simulates use errors as follows: Slip errors are a change of the order of execution of normative user actions; Lapses are an elimination of an action; and Mistakes are a change of a value in a form. While these formulations are

[1] Available at `http://gitlab.inesctec.pt/jccampos/ise-public-builds`.

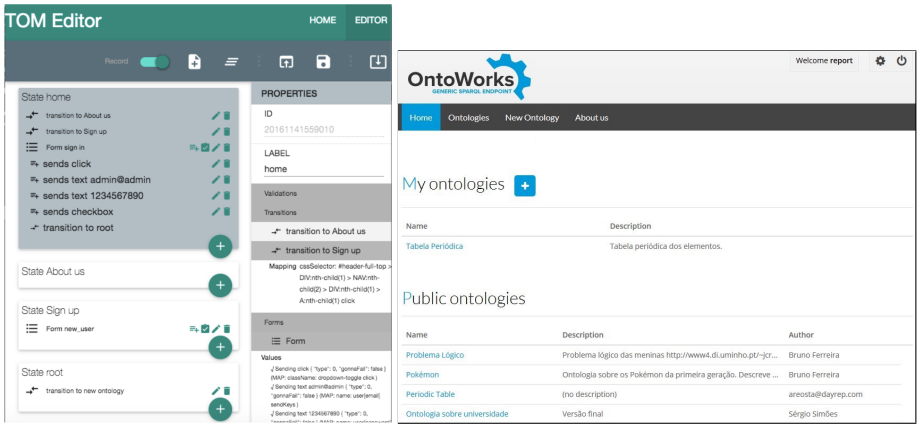

Fig. 2. TOM Web editor (left) and OntoWorks (right)

rather simple, they are sufficient for assessing the utility of the mutation approach used in the TOM framework.

3 Demonstrative Example

This section presents an example use of TOM to test OntoWorks, a Web application supporting online editing and querying of ontologies (see Figure 2, right). Features include: visualization and execution of queries; loading, removing, editing and visualization of ontologies; association of queries to ontologies.

Building the model. The TOM Editor was used to aid build the system model. The final model consisted of 15 states and 24 transitions. A screen-shot of the editor while building the model is in Figure 2: it shows the “home” state, which includes two transitions and a login form. Transitions for the home state lead to states “About us” and “Sign up”. Modeling the OntoWorks system in the TOM Editor took about 5 hours, a significant reduction compared to a previous manual modeling effort which took 27 hours. The bottleneck of the manual modeling effort was the cost of mapping each element of the state machine to the web page. In the current model of the system there are 102 such mappings, mostly obtained automatically. After constructing the model, validation checks were added to the states. This was manually done with support from the tool (e.g., to identify elements in the page). In total, 61 validation checks were defined, 57 when entering the web page and 4 when leaving. The "displayed?" validation was the most used (27 times). Three user-defined mutations were configured, which are specifically targeted at Web applications: pressing the back button; refreshing the page; double-clicking a user interface element.

Generation of test cases. The predefined *AllDirectedPaths* algorithm from JGraphT was used to traverse the graph. To ensure that the algorithm traverses all nodes and edges at least once, the following constraints were defined: the number of visits to nodes is at most 1, and the number of visits to edges is at most 2. A total of 273 paths were

generated, which are automatically converted into concrete test cases for OntoWorks using the exporter for Selenium WebDriver. Starting from these paths, TOM generated 2,730 additional test cases based on the identified mutation strategies. The test cases were run in Google Chrome. In case of error, a screen-shot of the web page was saved.

Results. A total of 935 step failures were obtained during the execution of the tests. The tests highlighted a latent implementation problem in the OntoWorks application (a same identifier was used twice in the same page, when it should have been unique), and gave us important insights on how to improve the generation and execution of test cases. These aspects are now discussed.

Positive and negative tests. Careful analysis of the test results revealed that several test failures were actually desired behavior of the system. For example, swapping the order of input in the user name and password fields prevents the login process. This indicates that we need to introduce the notions of "positive" and "negative" tests (i.e., tests that should be considered as passed if the interaction succeeds vs. tests that should be considered as passed if the interaction fails). Whether it will be possible to automatically categorize mutated tests into positive/negative tests needs to be explored.

Cascading errors. While our main goal was to detect as many errors as possible in a single test, we observed that a failed step in a test case tends to propagate to the remainder steps of the test case, causing the subsequent steps to fail too. The three steps that failed in the non-mutated tests category are an example of this. The problem happens in the login form: the user name input field is being reported as not visible. This error occurs because Selenium is attempting to populate a form field that is hidden at runtime. Manual inspection of the form, however, showed the field visible on the form. After inspection of the code it was realized that there were two elements with the same identifier in the same page, when they should have been unique. Therefore the failure in the test highlighted a latent implementation error. Subsequent failures in the test where due to the fact that the login process had failed, and not because of problems with the SUT. A redesign of the test case generation module is currently under way, that flags a test as failed as soon as a step in the sequence fails.

4 Conclusion

This paper described the TOM framework, which aims at supporting the model-based testing of interactive computing system. The architecture and main functionalities of the framework were introduced, including all steps necessary for the creation of the user interface model and the generation and execution of test cases. A layer supporting MBT of Web application was also developed and an example illustrating its application was described. The example makes use of TOM Editor, a model editor for web applications. The example application allowed us to identify a number of lines for future work: from the need to better consider the role of mutations in the test case generation process, to fine-tuning the executable test cases generation process. The definition of coverage criteria is also a topic for future work. The framework was developed in a modular and structured way, allowing the addition of new features. We plan to explore further the integration with task modeling tool-sets such as HAMSTERS [2], taking advantage of

information task models might have about use error to improve the test cases generation and mutation processes (see [5]). Other extensions under development include interface modules for Alloy [8] and the PVSio-web [10] prototyping tool.

Acknowledgments. Work financed by the ERDF (European Regional Development Fund) through the COMPETE 2020 Programme, and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826.

References

1. D. Amalfitano, A.R. Fasolino, P. Tramontana, B.D. Ta, and A.M. Memon. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.
2. E. Barboni, J.-F. Ladry, D. Navarre, P. Palanque, and M. Winckler. Beyond modelling: An integrated environment supporting co-execution of tasks and systems models. In *Proc. EICS'10*, pages 165–174. ACM, 2010.
3. A. Barbosa, A.C. Paiva, and J.C. Campos. Test case generation from mutated task models. In *Proc. EICS '11*, pages 175–184. ACM, 2011.
4. J. C. Campos and M. D. Harrison. Interaction engineering using the IVY tool. In *Proc. EICS'09*, pages 35–44, New York, NY, USA, 2009. ACM.
5. J.C. Campos, C. Fayollas, C. Martinie, D. Navarre, P. Palanque, and M. Pinto. Systematic automation of scenario-based testing of user interfaces. In *Proc. EICS'16*, pages 138–148. ACM, 2016.
6. A. Gimblett and H. Thimbleby. User interface model discovery: Towards a generic approach. In *Proc. EICS'10*, pages 145–154. ACM, 2010.
7. International Organization for Standardization. ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) - part 11: guidance on usability. *International Organization for Standardization*, 1998(2):28, 1998.
8. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
9. V. Lelli, A. Blouin, B. Baudry, and F. Coulon. On model-based testing advanced GUIs. In *Proc. 2015 IEEE 8th Intl. Conf. Software Testing, Verification and Validation Workshops (ICSTW), 11th Workshop on Advances in Model Based Testing (A-MOST)*. IEEE, 2015.
10. P. Masci, P. Oladimeji, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby. PVSio-web 2.0: Joining PVS to HCI. In *Computer Aided Verification*, volume 9206 of *Lecture Notes in Computer Science*, pages 470–478. Springer, 2015.
11. A.M. Memon. *A Comprehensive Framework For Testing Graphical User Interfaces*. PhD thesis, University of Pittsburgh, 2001.
12. R. Moreira and A.C. Paiva. PBGT Tool: An integrated modeling and testing environment for pattern-based GUI testing. In *Proc. ASE 2014*, pages 863–866. ACM, 2014.
13. I.C. Morgado and A.C. Paiva. The iMPAcT tool: Testing ui patterns on mobile applications. In *Proc. ASE 2015*, pages 876–881, 2015.
14. B. Nguyen, B. Robbins, I. Banerjee, and A. Memon. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering*, 21(1), 2014.
15. A.C. Paiva. *Automated Specification-Based Testing of Graphical User Interfaces*. PhD thesis, Engineering Faculty of Porto University, Dep. of Electrical and Computer Engineering, 2007.
16. J. Reason. *Human Error*. Cambridge University Press, 1990.
17. J.L. Silva, J.C. Campos, and A.C. Paiva. Model-based user interface testing with Spec Explorer and ConcurTaskTrees. *Electronic Notes in Theor. Comp. Science*, 208:77–93, 2008.
18. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2007.
19. W3C. State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C Recommendation, September 2015.