Towards Model Construction Based on Test Cases and GUI Extraction

Antti Jääskeläinen(✉)

Department of Pervasive Computing, Tampere University of Technology, PO Box 553, 33101 Tampere, Finland
antti.m.jaaskelainen@tut.fi

Abstract. The adoption of model-based testing techniques is hindered by the difficulty of creating a test model. Various techniques to automate the modelling process have been proposed, based on software process artefacts or an existing product. This paper outlines a hybrid approach to model construction, based on two previously proposed methods. The presented approach combines information in pre-existing test cases with a model extracted from the graphical user interface of the product.

Keywords: Model extraction · Model-based testing · Software testing

1 Introduction

Model-based testing is a testing methodology that automates the generation of tests as well as their execution. In a typical approach, the tester first creates a formal model (such as a state machine) that depicts the behaviour of the system under test (SUT). The model is then explored by an automated tool in order to generate a sequence of actions to be used as a test. Models can also be used to otherwise support the testing process, such as in inspections.

A significant drawback of model-based testing is the skill and effort required in modelling. Creating a model that covers all the relevant aspects of the SUT, does so correctly, and is otherwise suitable for test generation, is no small task.

Various methods for easing or partially automating the modelling process have been proposed. Models can be generated from different artefacts of the software process, or the artefacts used directly as test models. Suitable candidates include specifications [10] and pre-existing test cases [9,14]. Alternatively, models can be extracted from an existing product, either the source code [4,13,15], the structure and functionality of the graphical user interface (GUI) [1,6,11,12], or other known behaviour [7,8]. Many of these methods also use the results of the generated tests to further hone the model.

This paper outlines a hybrid model construction method based on test cases and information extracted from the GUI. Information on the correct behaviour of the SUT in specific states is found in the test cases, and the states can be combined based on information gleaned from the GUI. In this way, weaknesses of

F. Wotawa et al. (Eds.): ICTSS 2016, LNCS 9976, pp. 225–230, 2016.
DOI: 10.1007/978-3-319-47443-4_15

each method can be compensated for with the strengths of the other. Hopefully, the new method could reduce the effort required to produce a useful test model.

The rest of the paper is structured as follows: Sect. 2 presents the two model construction methods that act as the basis for the new approach, which is outlined in Sect. 3. Section 4 considers the potential benefits of the approach. Finally, Sect. 5 concludes the paper with a more general discussion.

2 Automated Model Construction

The approach of this paper builds on two previously presented methods for constructing models to describe the SUT. The first is based on combining test cases into a model, the second on examining the GUI of the system.

2.1 Synthesis from Test Cases

The model synthesis process proposed in [9] is based on pre-existing test cases that are linear sequences of automatically executable steps (*keywords*), and consists of five phases: First, keywords used in test cases are identified and classified. Second, part of the information in the test cases (such as input data) is separated into variables. Third, an initialization sequence for the test model is prepared. Fourth, important states in the test cases are identified manually. Finally, the actual merging of the test cases takes place.

In the merging phase, the linear state sequences of the test cases are combined into a more complex model by merging some of the states with each other. The previously identified common states in the test cases are trivially merged. However, states in different test cases may also be combined if they are reached by the same sequence of keywords, which suggests that the cases are in fact handling the same part of the SUT. Separating input data into variables allows states to be combined even if the stored inputs would actually leave the system in different states, as the data can be combined back into the model afterward.

Although this method works, it relies on a significant amount of manual effort. In particular, the separation of variables requires significant work and skill [9]. Also, the tester has to confirm the merges between the test cases manually, as the state sequence method may generate false positives [9]. Thus, the practicality of the method as presented is questionable.

2.2 Extraction from GUI

As an example of methods that extract a model out of a GUI we consider Murphy, a tool that examines the GUI of an application, tries out different functions, and builds a model to describe its observations [1]. It can use various methods to access the GUI, or *crawl* through it. The publicly available version [5] provides crawlers that use Windows APIs or cycle through GUI controls with the tabulator key. The constructed model is a graph with a node for each observable state of the application.

Murphy starts out at desktop, and launches the application with a predefined command. These two application states, not running and just launched, form the two first nodes in the constructed graph. In each GUI state, Murphy maps out the controls found in it. Then it proceeds to try out a control, such as clicking a button, and considers the resulting state of the GUI. A given state of the application is identified by the controls that can be found in its GUI, while ignoring the data such as the contents of text labels. Based on these, Murphy will either create a transition to an existing node or add a new node to the graph. Extraction can be performed in multiple runs starting from the desktop, with the results combined into a single graph.

Visual inspection of the extracted models can be very helpful in finding errors, and the models can support manual testing of the application [2]. They can also be used for automated regression testing by comparing the extracted models between different versions of the application or using an extracted sequence as a smoke test for the next version [2]. That said, their usefulness for test generation is limited: they can make no difference between an erroneous feature and a correct one, and contain no verifications of the system state beyond finding the expected controls in the GUI. Also, specific input data has to be added into the extraction script manually, as it cannot be inferred from the GUI.

3 Combined Methodology

Both approaches described above have their drawbacks. In the test case synthesis method, the test cases provide detailed information of what can and should happen in different situations, but constructing a model out of them is difficult. In the GUI extraction method, a graph to describe the SUT can be constructed with little manual effort, but its understanding of the SUT is limited. But what if the two were combined?

If we have a ready set of test cases when we begin the extraction process, then we can track the actions taken in the GUI within the cases. For each node of the constructed graph, we will have a set of test cases that reach the corresponding SUT state at a specific point of their execution. Then, we can examine the next steps in those cases for information on the current state of the SUT and the actions available in it. The process could work as follows:

1. Before extraction, establish a correspondence between the keywords within the test cases and actions supported by the extraction crawler. This is trivial if the two use the same mechanism for accessing the GUI, and should be doable with any mechanisms that understand the structure of a GUI. Also, make note of any keywords used to verify the state of the SUT without changing it.
2. At the beginning of an extraction run, start with the full set of test cases at their initial states. When the extractor performs an action, examine the cases to see if they would execute the corresponding keyword next, skipping past any keywords that do not change the state of the SUT. Advance these cases past that keyword, and discard the other cases in the set. Make a note that this node can be reached by the remaining test cases at their current stage.

3. At each node, examine the test cases that can reach it. Any verifications performed by the test cases at this node can be added into the model. Also, the next keywords in the cases should be executable in the GUI, even if the crawler fails to find a corresponding control. In particular, test data in the cases, such as the parameter of a *type text* keyword, can be used as an input.

This process produces a model that incorporates and combines both the information extractable from the SUT and that present in the test cases. It may even contain functionality present in neither, if the test cases provide the crawler access to a part of the GUI it could not reach on its own.

4 Potential Applications

Combining the test case synthesis and GUI extraction methods as described above offers several potential benefits. Either of the test cases or the extracted model can be used to support a testing approach based on the other. The resulting model can also act as a basis for a manually maintained test model.

If the testing process is based on test cases, their coverage can be increased with the information extracted from the GUI. With the extracted information, it is possible to tell when two test cases reach the same SUT state, or when a test case loops back into a state it has already visited. By using the model for test generation, it is possible to reach a state by a keyword sequence taken from one test case and continue with a sequence from another, even if such a combination does not occur in any of the original cases. A model that loops back into itself at several points can be particularly useful in robustness testing: properly directed, a test run can continue indefinitely without simply repeating a single sequence of keywords over and over.

Conversely, in a testing process based on the extracted model, the test cases can improve the quality of that model. They can improve model coverage by supplying performable actions that cannot be identified in the GUI, and in particular by providing realistic test data. Also, the verifications in the test cases improve the ability of the model to detect errors.

Finally, the constructed model can support a move to proper model-based testing, where the tests are generated from the model. Creating a test model can be a daunting task, especially if a fairly complete product already exists, so that the model cannot be developed incrementally as new features are added. In this situation, a method for automatically constructing a preliminary model can be helpful, even though some augmentation and refactoring is likely to be required. A model extracted purely from the GUI can already be useful here, but the increased coverage and verifications added by the test cases can take this support further.

5 Discussion

The previous sections have outlined a method for constructing models based on information in test cases and the GUI of the SUT. The resulting models can be used to improve test coverage in a testing approach based on test cases, or to better support static analysis and exploratory techniques. The method is language-independent, and only requires the SUT to have a GUI that can be handled by test automation. Obviously, it assumes the existence of some manually created test cases, and is thus primarily suited for testing approaches that will have those anyway. Practical experience will be required to see whether writing test cases specifically for this method could be worthwhile.

As presented here, the GUI extraction part of the process is based on the Murphy tool. However, there is nothing tool-specific in the approach itself, and other tools can be used, as well. The basic requirement is that the tool can distinguish different GUI events from each other, so that they can be matched with those occurring in the test cases. The test cases must naturally have a similar level of abstraction.

At this point, a prototype tool for the methodology is under development. The prototype can be used to estimate the practicality of the approach, although it will likely be unable to handle complex applications due to the limitations in the crawler component of the freely available version of Murphy [5]. For industrial use, integration with a professional quality test execution tool will be required.

There is likely also room for improvement in the methodology presented here. For example, it may be possible to factor out inputs from the test cases so that we could produce separate models for the control graph and saved data. Detecting the input data as identical output later on should be simple, but potential effects of data on the control graph could be more difficult to identify, and modified versions of the data impossible to recognize without domain knowledge. Likewise, it remains to be seen whether the Murphy approach of ignoring data in the GUI when identifying states is the best solution for the new method.

If the test cases have been created using action words and keywords [3], it might be possible to import these two tiers of abstraction into the constructed model. Presenting the model at a higher level of abstraction could make analysing it significantly easier, and produce a better basis for a full-fledged test model.

Acknowledgements. Funding by Ulla Tuominen Foundation is gratefully acknowledged.

References

1. Aho, P., Suarez, M., Kanstrén, T., Memon, A.M.: Industrial adoption of automatically extracted GUI models for testing. In: Chaudron, M., Genero, M., Abrahão, S., Pareto, L. (eds.) Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modelling (EESSMod 2013), CEUR-WS, vol. 1078, pp. 49–54. CEUR Workshop Proceedings, October 2013

2. Aho, P., Suarez, M., Kanstrén, T., Memon, A.M.: Murphy tools: utilizing extracted GUI models for industrial software testing. In: O'Conner, L. (ed.) Proceedings of the 7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2014), pp. 343–348. IEEE Computer Society, Los Alamitos (2014)
3. Buwalda, H.: Action figures. In: Software Testing and Quality Engineering Magazine, pp, 42–47, March/April 2003
4. Dallmeier, V., Knopp, N., Mallon, C., Hack, S., Zeller, A.: Generating test cases for specification mining. In: Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA 2010), pp. 85–96. ACM, New York, July 2010
5. F-Secure: GitHub - F-Secure/murphy (2014). https://github.com/F-Secure/murphy. Accessed June 2016
6. Grilo, A.M.P., Paiva, A.C.R., Faria, J.P.: Reverse engineering of GUI models for testing. In: Proceedings of the 5th Iberian Conference on Information Systems and Technologies (CISTI 2010), pp. 1–6. IEEE Computer Society, Los Alamitos, June 2010
7. Hagerer, A., Hungar, H., Niese, O., Steffen, B.: Model generation by moderated regular extrapolation. In: Kutsche, R.-D., Weber, H. (eds.) FASE 2002. LNCS, vol. 2306, pp. 80–95. Springer, Heidelberg (2002). doi:10.1007/3-540-45923-5_6
8. Hungar, H., Margaria, T., Steffen, B.: Test-based model generation for legacy systems. In: Proceedings of the 2003 International Test Conference (ICT 2003), vol. 2, pp. 150–159. IEEE Computer Society, Los Alamitos, September–October 2003
9. Jääskeläinen, A., Kervinen, A., Katara, M., Valmari, A., Virtanen, H.: Synthesizing test models from test cases. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 179–193. Springer, Heidelberg (2009). doi:10.1007/978-3-642-01702-5_18
10. Ma, C., Du, C., Zhang, T., Hu, F., Cai, X.: WSDL-based automated test data generation for web service. In: Kawada, S. (ed.) Proceedings of the International Conference on Computer Science and Software Engineering (CSSE 2008), pp. 731–737. IEEE Computer Society, Los Alamitos (2008)
11. Memon, A., Banerjee, I., Nagarajan, A.: GUI ripping: reverse engineering of graphical user interfaces for testing. In: van Deursen, A., Stroulia, E., Storey, M.A.D. (eds.) Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003), pp. 260–269. IEEE Computer Society, Los Alamitos (2003)
12. Memon, A.M.: An event-flow model of GUI-based applications for testing. Softw. Test. Verif. Reliab. (STVR) **17**(3), 137–157 (2007)
13. Silva, J.C., Silva, C., Gonçalo, R.D., Saraiva, J., Campos, J.C.: The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In: Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010), pp. 181–186. ACM, New York, June 2010
14. Xie, T., Notkin, D.: Mutually enhancing test generation and specification inference. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 60–69. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24617-6_5
15. Yang, W., Prasad, M.R., Xie, T.: A grey-box approach for automated GUI-model generation of mobile applications. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 250–265. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37057-1_19