The Synergy Between User Experience Design and Software Testing

A.P. van der Meer, R. Kherrazi, N. Noroozi(✉), and A. Wierda

Nspyre B.V., Eindhoven, The Netherlands
neda.noroozi@nspyre.nl

Abstract. Formal methods and testing are two important approaches that assist in the development of high quality software. Model-based testing (MBT) is a systematic approach to testing where using formal models enables automatic generation of test cases and test oracle. Although the results of applying MBT in practice are promising, creating formal models is an obstacle for wide-spread use of MBT in industry. In this paper we address how the cooperation between testers and user experience designers can help with the overall challenge of applying MBT. We present a test automation approach based on Task Models and Microsoft Spec Explorer model-based testing tool to improve software testing. Task Model is a formal model to specify the high-level interaction between the user and the graphical user interface (GUI). We developed a tool, called UXSpec, to convert Task Models to the input models of Spec Explorer, allowing us to do functional testing with little modeling effort, due to usage of already existing models. We demonstrate this by applying our approach to a case study.

Keywords: Model-based testing · Spec Explorer · ConcurTaskTrees

1 Introduction

The speed of software development increases steadily. As a consequence the challenges that each discipline faces increase. For User eXperience (UX) designers a challenge is to ensure that UX designs get implemented correctly, including aspects of performance, exception handling, etc. For software testers the challenge is to build a full understanding of what the correct system is in a limited amount of time. On a high level the solution we have found is to reuse the requirements efforts from UX as input for model-based testing to improve overall development speed and testing quality. Using this approach allows us to benefit from all powerful features of MBT tools without facing issues related to creation of complex test models.

UX work contributes to requirements creation by adding the user perspective; who is the user, what are the tasks that (s)he will use the system for? A tool to create this understanding is making a task analysis. The widely used notation for this is Concurtasktree (CTT) [10]. We found that we can reuse CTT models,

D. Bianculli et al. (Eds.): SEFM 2015 Workshops, LNCS 9509, pp. 11–21, 2015.
DOI: 10.1007/978-3-662-49224-6_2

which capture the UI requirements, as basis for model-based testing of the final implementation. For this we have developed the UXSpec tool which converts CTT models to the test models used by Microsoft Spec Explorer tool [1].

Furthermore we have found that the discussions between testers and UX-ers is useful to better coordinate usability testing effort, e.g. what are the most critical scenarios, and which user tasks should be included in a usability test. This qualitative testing complements the testing efforts aimed at coverage and completeness and improves the overall system quality.

In this paper, we first introduce Task Models and Spec Explorer in Sect. 2. We describe some technical details of the implementation of our approach in Sect. 3. In Sect. 4, we report on a case study to demonstrate the feasibility of our tool chain and present some empirical results. We discuss related work in Sect. 5, and finally we write the conclusions in Sect. 6.

2 Preliminaries

2.1 Task Models

Task Models are useful in designing and developing interactive systems. They describe the logical activities (tasks) that have to be carried out in order to reach the users goals [11]. A widely used notation for Task Models is ConcurTaskTtree (CTT) [10], which we use in this paper. Four different types of tasks which are supported by CTT models are as follows:

- *interaction task* represent events initiated by a user of the systems
- *application tasks* represent system responses
- *user tasks* represent decision points on the user side
- *abstract tasks* which are further subdivided in other tasks

Connections between tasks are annotated with the temporal operators that describes the dependencies between tasks. Providing a rich set of operators, a set of temporal relationships between tasks such as enabling, disabling, choice, and synchronization are formally defined. A part of the CTT model of a car reservation which will be used as the case study in Sect. 4 is shown in Fig. 1. The sub-tree depicted in Fig. 1 shows the task model for adding a new car to the car reservation system. This CTT model consists of six *interaction* tasks including navigation to the desired interface and providing the required data, and three *application* tasks showing the responses of the system. Semantic of the model is defined by the possible exploration of the model which is governed by the temporal operators. The task of adding a car is decomposed to two interaction tasks: first executing task 'Navigate to Cars Mng' and then performing task 'Add Car X' (enabling operator: $\gg$). To navigate to the 'Cars' tab, the cars tab must first be selected (interaction task 'Click Cars tab') and then the system shows the Cars tab (application task 'Show Cars tab'). A new car is now added by selecting the add car option (interaction task 'Select Add Car') which yields a pop-up window is opened (application task 'Show Pop-up Win'). At this

point the user can enter the information of the new car by entering two possible data (choice operator: []): Select the car model or Select the station. Task 'Enter Car Information' is an iterative task (iterative operator: *), meaning that it can be repeatedly executed. Once, the user confirm the entered information, (s)he cannot change those information anymore (disabling operator: [>). Eventually, the updated list of the cars is shown to the user (application task: Update Car List). For a complete list of the operators and their meaning see [11].

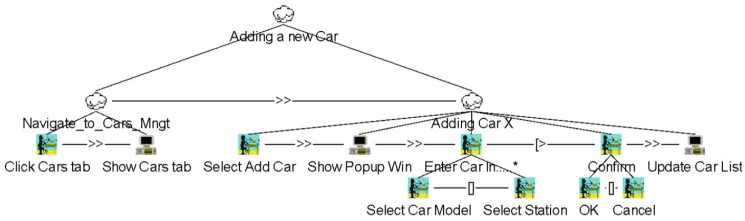

Fig. 1. Fragment of a car reservation system ConcurTaskTree model (in MARIAE tool)

Although it is not visible in Fig. 1, the task model is enriched with some constraints to ensure the availability of certain actions at different states of the system. These constraints are expressed as pre/post conditions of an atomic task. For example, the task named select station in Fig. 1 has a post-condition which guarantees that a station is always selected as one of the requirements of the system. To this end, a string variable named station with the initial value empty is defined in the task. The post-condition, then, is presented by the expression station != empty where station variable shows the name of selected station. A similar post-condition is added to task 'select car model' to ensure that the mandatory field of car model is always set to a non-empty value.

2.2 Spec Explorer

Spec Explorer [1] is an extension to Microsoft Visual Studio intended to provide support for MBT. In Spec Explorer, we define a model that describes the expected behavior of the system under test together with a configuration file in which we describe parameters of test cases. The modeling language used in Spec Explorer is an extension to C# with modeling annotation and construction like pre/post condition, final states, variables, etc. The model of a system described by the Spec Explorer modeling language is a possibly infinite state transition system in which the states comprise the possible states of the system variables and transitions between the states are modeled by methods annotated as *Action*. There are two different types of actions in Spec Explorer: controllable and observable actions. Controllable actions defines actions that are under the control of the user (inputs of the system under test), and observable actions describes the (asynchronous) responses of the system. Once the model of a system is complete

in Spec Explorer, different predefined test strategies can be applied to generate test cases, which can be executed directly on the implemented system. Each path in the state space of the defined state transition system represents a possible test, a sequence of controllable and observable actions that the system under test has to be able to follow. More information on Spec Explorer can be found in [9].

3 Using Task Models in Model-Based Testing

In order to generate tests for a system in Spec Explorer, we need a model that describes the expected behavior and a script file that defines test scenarios, i.e. the kind of tests to be executed. Creating behavioral models is a non-trivial process which typically requires considerable time and efforts in MBT approaches. When we design a system by using CTT models, we already have a model that describes the desired behavior of the system under test. This suggests that if we reuse this model, we will make testing with Spec Explorer easier and cheaper.

Since CTT models cannot be directly used in Spec Explorer, we need to create a Spec Explorer model based on a given Task model in a way that does not affect the semantic of the given CTT model. From a global perspective, we observe that both CTT model and Spec Explorer are fundamentally state-transition-system-based formalisms. This means a system at all times has a well-defined state, which changes in response to triggers received from the environment. In particular, this implies that we can consider a Spec Explorer model compliant to a CTT model if the state machines involved are equivalent in behavior. More precise, we need to make sure that the tests that are constructed by Spec Explorer contain all possible sequences of event triggers that are legal in the CTT model state machine. On the other hand, illegal events should never occur in tests, because the behavior of the component is undefined in such cases, which means that the test can never be failed or passed.

The state transition system of a CTT model can be extracted by identifying its states and the transitions between them. This can be done by simulating the execution of task models: each set of enabled tasks at a same time, called Presentation Task Sets (PTS), in the execution of the task model represents a state of the model, and the relation between tasks defines transitions between states. This step is automatically carried out in MARIAE tool.

In Spec Explorer, the model of the system under test is defined in one or more C# classes with methods which describe events that the system responds to or produces in the response to another event. By analyzing the effects of the methods on the state of the system, Spec Explorer can identify states and the transitions between them. Thus, to reconstruct a CTT model state machine in a Spec Explorer model, we have to create model classes that implement all possible events of the CTT model, in such a way that the resulting state space matches the one in the CTT model. The former is simply achieved by creating a method (defined as a controllable action) for every *interaction* task in CTT

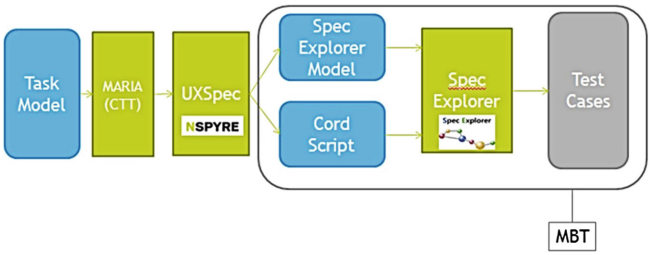

Fig. 2. UXSpec workflow and tool chain

models. We achieve the latter by constructing the model, based on an explicit state machine pattern, thus ensuring that all states are explicitly present in the model. In the same way, all events are explicitly covered in methods, ensuring the full coverage of all transitions of CTT models. Furthermore, application of tasks in the task model are translated as probe methods in the Spec Explorer model to check if the effect of user interaction is as expected. The generated Spec Explorer model can be later enriched by further details to each state or/and by defining data constraints in the model which are not available in CTT models. To do the translation from CTT models to Spec Explorer models automatically, we developed a tool, which is named UXSpec. The general work flow around this tool chain is shown in Fig. 2. The UXSpec tool takes the CTT models as input and combines them with some configuration data that are not presented in CTT models, for example how the user interface implementation can be started and stopped. The output of the tool is a Spec Explorer model that can then be used to generate test cases.

UXSpec uses a model transformation in QVTo [5,8], an Eclipse Modeling Framework (EMF) implementation of QVT Operational, to carry out the transformation of task models in CTT notation to models used by Spec Explorer. Because QVTo is based on EMF [3], we translate CTT models into EMF model first. For this, we used an existing tool based on the XML schema provided by the HIIS Laboratory for MARIAE [15], describing the structure of CTT models. This tool was developed by Nspyre as part of an earlier project. In the same project, a transformation was developed that abstracts from format-specific features of CTT models to a generic, more abstract representation. We use this representation as a basis for further processing.

The next step is to generate the C# model file and the Cord Script file, which is implemented by means of two QVTo transformations. The first one generates an EMF C# model based on a C# meta-model created by the MoDisco [4] project. The second generates an EMF Cord Script model based on a Cord Script meta-model of our own design. Because these are both in EMF format, we then have to use templates, in our case based on the Acceleo [2] template engine, to create the textual representations that can be used by Spec Explorer. These templates are generic, in the sense that they can be used for all EMF C# and Cord Script models that use our meta-models.

4 Case Study

In this section we demonstrate the feasibility of our approach with an example. The application being used for this purpose is a car reservation system which is based on a large industrial system co-developed by Nspyre. The car reservation system in this section is developed for car rental agencies for booking and managing rental cars. It consists of various user interfaces and each of them has several GUI components. We focus on the functionality of adding a new car to the system. A car is added to the system by identifying its model and the station to which it is assigned as the mandatory fields and its type as an optional field.

To evaluate our approach, we tested the above functionality of the car reservation system with two different techniques: Traditional test automation approach (capture/replay) and (conventional) model-based testing. Capture/replay tools are widely adopted for automatic test execution in which a test is designed and executed for the first time by a human and then a test executor executes scripted tests that record the human interactions with the system under test. For this purpose, we designed test scenarios by using decision table technique. Afterwards, scripted tests of each logical test case are generated by hand. Regarding the system under test was a windows application in this case study, we developed an automatic test executor based on UI Automation framework [12] which translated abstract user interactions of scripted tests to actual UI events, and vice versa.

In the second approach, test cases are automatically generated in Spec Explorer from test models that are manually created. As the common practice in MBT, in order to execute the generated test cases on the system under test, we developed an adapter based on UI Automation framework to connect to the system under test. Finally, we test the system by reusing the CTT model depicted in Fig. 1 and using UXSpec tool to automatically translate the CTT model to a test model in Spec Explorer.

In the remainder of this section, we first explain all steps of the process of automatic generation of test cases in our approach by using UXSpec tool. Afterwards, we compare our approach with the other two approaches.

4.1 Testing with UXSpec Tool

Having the task model, the state transition system of the CTT model is automatically generated by MARIAE tool. UXSpec tool starts with the generated state transition system. However, all information of CTT models like pre/post condition of tasks, is not available in the transition system. Therefore, UXSpec use the actual CTT model to extract the necessary information. The output of UXSpec is the preliminary Spec Explorer model in C# format together with a script file, which represents the intended behavior of the system. Figure 3 shows the graphical representation of the Spec Explorer model generated by UXSpec from the task model in Fig. 1. The gray state shows the initial state and each arc shows an enabled interaction task at each state which is translated to a method in the C# model. For example consider the initial state in the CTT model at

which the user can only click on Cars tab (interaction task 'click cars tab'). This interaction task translated in Spec Explorer model as controllable method 'P2_Click_Cars_Tab()'. Moreover, to check the effect of tasks (post condition), probe methods are created in the generated Spec Explorer model. For instance, the interaction task 'select station' in Fig. 1 has a post-condition which guarantees that a station is always selected. This post-condition is defined in the CTT model by the expression "station != empty" where "station" variable shows the name of selected station. This post-condition is checked by probe method 'getStation()' in the Spec Explorer model. The generated model can be later enriched by further details to each state or/and by defining data constraints in C# model which are not available in task models. For instance, we restricted the set of possible stations and car models to two certain sets of values. These modifications took a small amount of time, i.e. less than half a man-hour. Finally from the modified model, test cases are automatically generated by Spec Explorer. To enable automatic execution of generated test cases over the system under test, we reused the developed adapter from the previous model-based testing project.

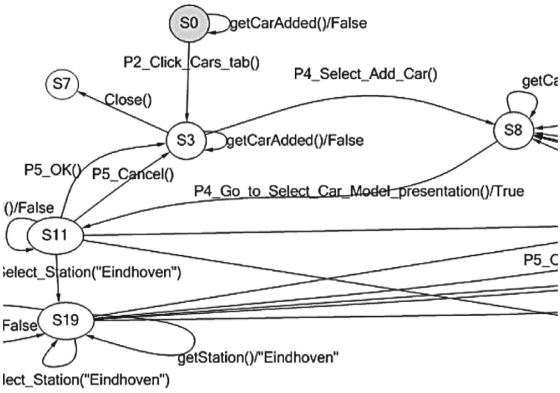

Fig. 3. A part of generated Spec Explorer model of the car reservation system

4.2 Results

Based on the case study, we have drawn several conclusions on the approaches discussed in this paper

Modeling. Task Models concentrate on activities that a user intends to perform over an interactive application together with temporal relationships between them. Task Models are created in early phases of software development by user experience designer. In cooperation with software architecture, these models are translated into detailed design in later phases. C# models created in Spec Explorer are intended to use for test case generation. They model some parts of

the functionalities of the system under test that are relevant in testing. These models are created by testers to be used only for testing the actual implementation. In our approach, UXSpec reuses existing CTT models to automatically generate C# models in Spec Explorer.

Technique. Using the formally defined semantics of temporal relationships between tasks, a designer is able to simulate the flow of the activities of a CTT model before designing the user interface to check if users goal is supported. In contrast, conventional MBT creates tests based on manually created models. UXSpec creates automatically primarily test models from existing CTT models.

Effort. Task Models are constructed by user experience designer by focusing on user interactions and abstracting from design and implementation details. In contrast, test models are created by testers to cover only those parts of the functionalities of system which are relevant to testing. UXSpec creates test models by reusing existing task models. Thus, it decreases effort needed for testing.

Requirement Coverage of Generated Test Cases. In order to give an indication of the efficiency of our approach; we look at the number of test cases generated by each approach in a same amount of time. In the first approach where the manual scripted tests are used, the number of generated test cases in one hour is 2. This number of test cases partially covered the required test scenarios. In the second approach, the test model of the system was manually developed in Spec Explorer in one hour. Afterwards, 13 test cases were generated in some milliseconds, which cover all possible scenarios. In our approach, the creation of the task model has taken about half an hour. Reusing the task model, UXSpec automatically generates a preliminary model in Spec Explorer. After customizing the generated model, 13 test cases were generated which have a same coverage as those generated in the pure MBT approach. Therefore, by using UXSpec we can reduce time and cost of modeling while preserving the requirement coverage.

5 Related Work

In this paper, we focus on improving and accelerating the use of model-based testing for applications interacted via their user interfaces. There are several works on using models in testing user interfaces [13–18]. Challenges of using model-based testing in GUI testing are comprehensively studied in [16,18]. A pure model-based testing approach is presented in [16]. Similarly, [15] reported on a model-based GUI testing approach in which Uppaal models are used as test models. In contrast to our approach, in both approaches in [15,16] test models need to be manually developed. Creating test models is not a trivial process and most of the time is time-consuming, particularly when the system under test is large. To overcome this problem in [17], a model-based tool, GUITAR, is developed which automatically creates test models as an event-flow graphs by extracting structures and API calls. This approach can be used only in the final phases of software development, when the system under test is developed.

However, using CTT models enables us to generate test cases in the early phase even before graphical user interfaces of the system with their implementation details are designed and implemented. Analogous to our approach in [13,14], CTT models are used as input for generating test models of graphical user interfaces. Instead of testing the desired behaviors modeled in task models, the focus of presented approach in [14] is on testing unexpected and undefined behavior by generating task mutations based on a classification of the user errors. The most similar work to the approach presented in this paper is [13], in which customized concureTaskTree models are automatically converted to C# models in Spec Explorer. CTT models used in [13] are enriched by some implementation details that are used later in development of the adapter and for checking the effect of interaction tasks. Including some implementation details in test models restricted the approach in [13] to testing Windows Form applications. Moreover, in [13] only 'interaction' tasks are taken in the transformation. But by using standard CTT models, our approach has general applications and it is not limited to a specific type of application and platform. Furthermore, instead of customizing CTT models with some data tag to express the effect of interaction tasks, the approach taken in [13], we use the post-conditions defined for atomic tasks. The post-conditions are then translated as the probe method in Spec Explorer models. Therefore, in contrast with [13], no customization of CTT models is needed in our approach.

6 Conclusions

In this paper, we described an approach to automatic testing that is based on combining Task models and model-based testing. We showed that creating the models needed for MBT could be done in a way that builds further on material that is already created earlier in the development process. This reduces rework, improves the understanding for all people involved in the system development and most importantly increases the total software development speed. Task models become shared models that are understood by all disciplines in the project from user experience designers to system architects to developers to testers. Besides improving the communications in software development team, the consistency between requirements defined in the early phases of software development with test cases executed in the later phases is improved as well.

To use Task models in model-based testing, we develop UXSpec tool which via some QVTo transformers translates task models from the MARIAE tool to test models used by Microsoft Spec Explorer tool. This enabled us to automatically create test cases based on a Task models. Although creating this custom connection takes a little bit of time we found that it significantly reduced time and efforts needed for test generation and maintenance. Moreover, the QVTo transformers of UXSpec support transition-based models as the target model. Therefore, the approach we demonstrated in this paper is not limited to a specific test tool and can be used with other MBT tools, such as NModel [6] or PyModel [7] as well.

To demonstrate the feasibility of our approach, we tested a simple part of a car reservation system with different testing techniques: the traditional test automation approach, and the conventional model-based testing in which test models are manually created. Although the scope of our case study is small, our obtained results are promising and showed reduction in time and effort needed for creating models in model-based testing. However, to strengthen our results and to have a better evaluation, we consider applying our approach on a real large industrial system in future.

References

1. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with Spec explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 39–76. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78917-8_2
2. The Eclipse Foundation: Acceleo. http://www.eclipse.org/acceleo/
3. The Eclipse Foundation: Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/modeling/emf/
4. The Eclipse Foundation: MoDisco Homepage. http://www.eclipse.org/MoDisco/
5. The Eclipse Foundation: QVTo. http://wiki.eclipse.org/QVTo
6. Microsoft: NModel. https://nmodel.codeplex.com/
7. Jacky, J: PyModel: model-based testing in Python. In: Proceedings of the 10th Python in Science Conference, pp. 43–48 (2011)
8. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Frame-Work 2.0, 2nd edn. Addison-Wesley Professional, Reading (2009)
9. Model-based Testing with SpecExplorer. http://research.microsoft.com/en-us/projects/specexplorer
10. Manca, M., Patern, F., Santoro, C., Spano, L.D.: Considering task pre-conditions in model-based user interface design and generation. In: Symposium on Engineering Interactive Computing Systems, pp. 149–154. ACM (2014)
11. Patern, F., Santoro, C., Spano, L.D., Raggett, D.: MBUI - Task Models, W3C Working Group Note 08 April 2014
12. Windows Automation API: UI Automation. http://msdn.microsoft.com/enus/library/windows/desktop/ee684009(v=vs.85).aspx
13. Silva, J.L., Campos, J.C., Paiva, A.C.R.: Model-based user interface testing with Spec explorer and ConcurTaskTrees. Electron. Notes Theor. Comput. Sci. **208**, 77–93 (2008). doi:10.1016/j.entcs.2008.03.108
14. Barbosa, A., Paiva, A.C.R., Campos, J.C.: Test case generation from mutated task models. In : Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2011), pp. 175–184. ACM (2011). doi:10.1145/1996461.1996516
15. Hjort, U.H., Illum, J., Larsen, K.G., Petersen, M.A., Skou, A.: Model-based GUI testing using Uppaal at Novo Nordisk. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 814–818. Springer, Heidelberg (2009). doi:10.1007/978-3-642-05089-3_53
16. Paiva, A.C.R.: Automated Specification-based Testing of Graphical User Interfaces, Ph.D. thesis, Faculty of Engineering, Porto University, Porto, Portugal (1997)

17. Nguyen, B., Robbins, B., Banerjee, I., Memon, A.: GUITAR: an innovative tool for AU-tomated testing of GUI-driven software. Autom. Softw. Eng. **21**, 65–105 (2013). doi:10.1007/s10515-013-0128-9
18. Alsmadi, I., Samarah, S., Saifan, A., AL Zamil, M.G.: Automatic model based methods to improve test effectiveness. Univ. J. Comput. Sci. Eng. Technol. **1**(1), 41–49 (2010)

http://www.springer.com/978-3-662-49223-9

Software Engineering and Formal Methods
SEFM 2015 Collocated Workshops: ATSE, HOFM, MoKMaSD, and VERY*SCART, York, UK, September 7-8, 2015. Revised Selected Papers
Bianculli, D.; Calinescu, R.; Rumpe, B. (Eds.)
2015, XXIX, 325 p. 88 illus. in color., Softcover
ISBN: 978-3-662-49223-9