Model-Based Test Instantiation for Applications with User Interfaces

Benedikt Hauptmann
Technische Universität München, Germany
benedikt.hauptmann@in.tum.de

ABSTRACT

Scripts for automated system tests often contain technical knowledge about the user interface (UI). This makes test scripts brittle and hard to maintain which leads to high maintenance costs. As a consequence, automation of system tests is often abandoned.
In this paper, we present the goals of our research project and discuss a model-driven approach to ease instantiation of tests on the system level. Tests are defined on an abstract, functional level, abstracting away UI usage. During test execution, abstract tests are enriched with UI information and executed against the system.

1. INTRODUCTION

Testing is a central activity for quality assurance. The system under test (SUT) is executed with the intention to find errors as well as to gain confidence that it works as intended.

System tests execute the whole system to check whether it fulfills its functional requirements. The focus is on the observable input/output behavior rather than on the structure or the internal state of the system or on timing aspects [1]. For systems with user interfaces (UI) this usually means that system tests are executed against the UI.

The execution of UI-based tests is easy to perform by humans. Their brainpower, experience and intuition facilitate them to interpret high level descriptions (e.g., *activate the human resource module*) without the need of detailed UI specific information.

If system tests have to be executed repeatedly, for example for regression testing, *test automation* can be very efficient [6, 9, 3]. To automate system tests, UI specific information has to be included in the *test scripts*. As such knowledge may change when the software evolves (e.g., a button is moved to a different dialog), automatically executable test scripts tend to be fragile and need to be maintained often [2, 18]. This causes considerable costs and the decision whether and when tests should be automated highly depends on the maintenance effort for the test scripts [6, 5]. The following problems motivate an efficient and flexible way of test instantiation and execution for systems with UIs.

1. *Maintenance of tests:* If it comes to changes of the UI, test scripts tend to be fragile because of their mixture of functional (e.g., providing a certain input value) and technical aspects (e.g., clicking on a specific button). This makes it difficult to adapt test scripts when the SUT changes.

2. *Reuse between tests:* The same functionality is normally tested with different inputs or in different variants. Furthermore, different tests may use the same parts of the user interface. A main obstacle for maintainable tests is redundancy, or put differently, the low level of information reuse within tests.

In this research project, we propose a model-driven approach to separate tests from UI related information. Tests are defined on the functional level, abstracting from UI interactions. During test execution, abstract tests are enriched with UI information and executed against the system.

The remainder of this paper is structured as follows. In Section 2, we introduce our conceptual idea to address the mentioned problems. In Section 3, we present the expected outcomes and contributions of our research project. In Section 4, related work is reviewed. In Section 5, a plan for the evaluation of this project is presented. The final Section describes the project's current status and gives an outlook on the next steps planned.

2. PROPOSED SOLUTION

To describe our solution, we introduce a model of a generic mediator acting as a broker between the user and the actual application. After that, we present our approach to a solution referring to this model.

2.1 Problem Analysis

To analyze the problems mentioned, we make use of a conceptual UI model introduced in [21] (see Figure 1). The model separates an interactive system into two logical parts: the *application* and a *mediator*. The application realizes the desired functionality of the system and is independent from any UI related concepts. The mediator is the intermediary

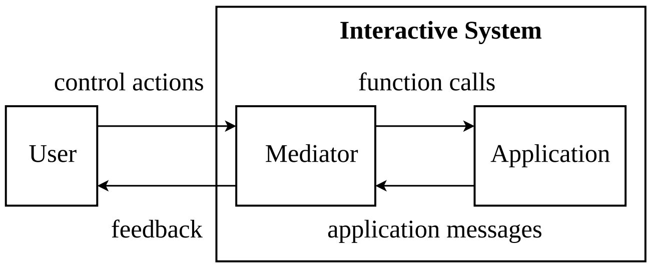

Figure 1: Conceptual UI Model (based on [21])

between the functionality and the user and forms the interface at the system boundary. It is built to be used by a certain type of user and therefore optimized for it. A system may also have several mediators for several types of users.

A telephone answering device, for example, typically has a UI containing a display and several buttons to be used in a physical way. The same machine may also be used remotely by voice control. Both UIs (mediators) will trigger the same functionality (the application) even though they have to be used in totally different ways.

In the sense of this model, an automated test is a special case of a user. However, the mediator is normally not optimized to be used by a test. An easy solution would be to extend the system with another mediator, optimized for automated testing (e.g., a special testing interface). This is in conflict with the paradigm of acceptance testing which is to test the complete system in a black box way including the mediator.

2.2 Approach

Since the SUT cannot be adapted for automated testing, a way to ease automated communication with the SUT has to be found.

Our approach aims to separate tests into functional and mediator specific concepts to improve their reuse and make them insusceptible to changes of the system's UI. We split tests into two artifacts:

1. Pure test logic (the actual *test cases*) which has no dependency to any information of the UI, and
2. the pure *UI knowledge* which is necessary to execute the tests (see *Test Cases* and *UI Model* in Figure 2).

For the latter, we suggest a descriptive model. To execute these UI independent test cases, we reconstruct the necessary information by using a generic *test adapter* which instantiates test cases using the UI information stored in the UI model for a given UI (see *Test Adapter* in Figure 2).

To handle the complexity of sophisticated UIs, we divide the test instantiation into several parts. We propose a proceeding which is oriented towards multi-layered *communication abstraction* [15] as in network communication stacks (e.g., the ISO/OSI reference model). We introduce several

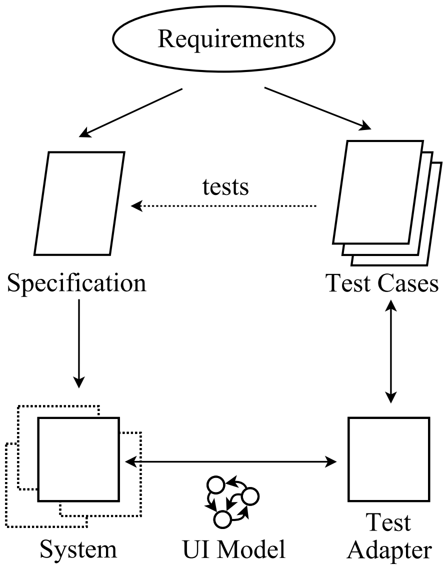

Figure 2: Overview of the approach.

abstraction layers stepwise reducing mediator specific details by abstracting the mediator's usage. Every layer reduces typical UI concepts (for example widgets and dialogs in graphical UIs or speech signals in voice UIs), and provides a more abstract view of the system's interface (see *System Interface*, *System Interface*$_{layer1}$, ... in Figure 3).

These virtual, more abstract SUT interfaces are the base for the next layers which will reduce further UI concepts. This is repeated until all UI related concepts are removed and a virtual, completely UI independent SUT interface has been created. Test cases are defined on this, most abstract layer using the interactions provided by this UI independent SUT interface. To execute tests, on each layer, models hold the extracted UI information. The adapter uses these models to concretize tests respectively abstract the system's response from one layer to another and finally executes the tests against the actual SUT (see ..., *Test*$_{layer2}$, *Test*$_{layer1}$, *Test* in Figure 3).

3. EXPECTED CONTRIBUTIONS

Based on our approach to separate tests from UI related information, we expect the following outcome of our research project:

1. *A method to describe abstract test cases.* We envision a description technique that largely uses natural language. However, as the abstract test cases will focus on input and output, tabular notations may be suitable for certain contexts as well. In order to reach our goal of enabling reuse, the language has to contain mechanisms to reuse test-fragments.
2. *Suitable UI models* containing the information used for concretizing abstract test. The formalism should be abstract enough to allow to intuitively model the aspects of the UI that are relevant for testing using

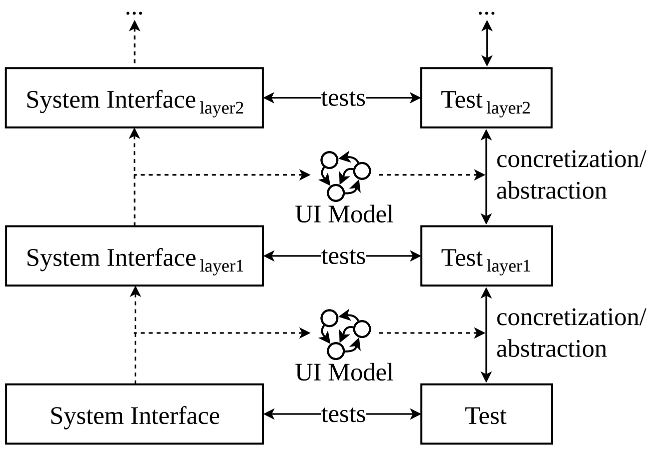

Figure 3: The introduced Approach

different UIs as well as powerful enough to allow modeling of sophisticated UIs and serve as an adapter to test-automation tools.

3. *A procedure that integrates the two artifacts above* in order to obtain executable test cases. In this procedure, abstract tests are therefore interpreted and enriched with information of the UI specification.

4. RELATED WORK

Improving maintainability of test scripts has been discussed repeatedly. Two popular concepts are *data-driven testing (DDT)* and *keyword-driven testing (KDT)* [6]. In the latter, to write abstract tests, action words are used which are mapped to test scripts. Our approach is similar to KDT but models the relevant parts of the UI instead of defining executable scripts. With this, we expect simpler maintenance and better reuse.

In [8], tool-support is used to ease the maintenance of test scripts for graphical UIs (GUIs). Differences between GUIs are automatically analyzed and the affected parts of the test scripts are detected. However, this approach still focuses on low level test scripts and does not exploit the full potential for test reuse.

Model-based testing (MBT) is concerned with testing software using models. However, the focus of most work is on test case generation, which we do not target. To generate test cases for GUIs, many approaches exist [12]. For example, [16, 13] use variants of finite state machines (FSMs) which are very detailed and complex to create and maintain. In our work, we want to keep the effort for the test designer as low as possible, by reducing the necessary models to just that information required for test execution.

Although it is acknowledged that the gap between abstract test cases and the system needs to be bridged [19], test instantiation in MBT is not covered in detail in most works. Most MBT approaches in literature are applied to systems with simple UIs, for example, embedded systems or chip cards. Mostly, there is a direct mapping between abstract actions and the code that implements that action. This is true, for example, for approaches that build on SpecExplorer [20, 13, 14]. Katara et al. [11] implement a layered approach for test case execution.

In [7] a model-driven approach which stepwise enriches tests with UI information is presented. However, since they focus only on infotainment systems of cars, they are limited in their field of application.

5. EVALUATION PLAN

To evaluate our research results, we conducted the following phases:

1. *Literature study:* Right now, we study related work in the area of MBT, automated test instantiation and execution, and *model-based UI design* (*MBUID*) [17, 4, 10] and evaluate existing tools for UI testing.

2. *Prototypical implementation and proof-of-concept:* To gain a better insight into suitable abstraction levels and to define a language for abstract tests, we build a prototypical implementation of our approach and apply it on a medium size open-source software. With this, we improve the UI modeling language as well as show the feasibility of our approach.

3. *Real world case study:* Once we have analyzed abstraction levels according with meta-models, we will perform a case study with an appropriate company. With this, we will improve the definition language for abstract tests as well as its integration in our UI modeling language. Furthermore, this will show the practicality of our approach.

6. CURRENT STATUS AND FUTURE WORK

To demonstrate the feasibility of our approach, we created a prototypical implementation of our approach. We built a test adapter acting as a generic abstraction/concretization framework which can be parameterized using UI models.

Borrowing from MBUID, we created a meta-model to create UI models for graphical UIs (GUIs). MBUID makes use of a set of interconnected models, putting down the different aspects of GUIs on various abstraction levels (by using for example dialog, navigation, and task models). Using these models, we perform a traceable, stepwise abstraction from UI information such as concrete UI widgets up to abstract interactions like *input*, *output*, or *action*.

As test object, we have chosen the open-source application Bugzilla[1], a web-based general-purpose bug tracking system. It is a grown system and many old versions are still available with which we can simulate an evolving system. Bugzilla started as a web application, however, there exists an Eclipse integration that offers (at least parts of) the same functionality. With this, we can validate the hypothesis that our approach is applicable to multi-frontend systems.

[1] `http://www.bugzilla.org`

As system specification, we used the official Bugzilla manual[2]. Even though the manual contains many GUI related details, the functional concepts are clearly recognizable. We modeled the relevant parts of both UIs and wrote several tests based on the functional specifications given by the official Bugzilla manual.

Based on the first experiences, we evaluated several abstraction levels for GUIs. We created a suitable meta-model based on the concepts from MBUID. Since the UI concepts of rich clients and web applications are alike, the abstractions we performed so far fit for both Bugzilla UIs equally well. We are able to run the same tests on both systems.

Based on our evaluation plan, the proof-of-concept has to be continued to improve the UI modeling language. The number of test cases has to be extended to create a reasonable coverage of both, Bugzilla's Web and Eclipse UI. Furthermore, to start a case study, an industry partner has to be found to apply our approach on a real world project. With this, we want to improve the test definition language, its integration in our UI modeling language. Furthermore, this allows to validate the practicality of our approach.

7. ACKNOWLEDGMENTS

The author thanks Maximilian Junker, Elmar Juergens, Stefan Wagner, Sebastian Eder and Andreas Vogelsang from Technische Universität München for their support and insightful comments.

References

[1] IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries. *IEEE Std 610*, 1991.

[2] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering (ICSE '05)*, 2005.

[3] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)*, 2007.

[4] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, N. Souchon, L. Bouillon, M. Florins, and J. Vanderdonckt. Plasticity of user interfaces: A revised reference framework. In *Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design*, 2002.

[5] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance.* Addison-Wesley, 1999.

[6] M. Fewster and D. Graham. *Software test automation: effective use of test execution tools.* Addison-Wesley, 1999.

[7] H. Grandy and S. Benz. Specification based testing of automotive human machine interfaces. In *GI Jahrestagung*, 2009.

[8] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, 2009.

[9] M. J. Harrold and A. Orso. Retesting software during development and maintenance. In *Proceedings of the Frontiers of Software Maintenance (FoSM '08)*, 2008.

[10] H. Hussmann, G. Meixner, and D. Zuehlke, editors. *Model-Driven Development of Advanced User Interfaces.* Springer, 2011.

[11] M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, and M. Satama. Towards deploying model-based testing with a domain-specific modeling approach. In *Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, 2006.

[12] A. M. Memon and B. N. Nguyen. Advances in automated model-based system testing of software applications with a GUI front-end. In M. V. Zelkowitz, editor, *Advances in Computers*, volume 80. 2010.

[13] A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Modeling and testing hierarchical guis. In *Proceedings of the 12th International Workshop on Abstract State Machines*, 2005.

[14] A. Pimenta. *Automated Specification Based Testing of Graphical User Interfaces.* PhD thesis, Engineering Faculty of Porto University, Department of Electrical and Computer Engineering, 2006.

[15] W. Prenninger and A. Pretschner. Abstractions for model-based testing. *Electron. Notes Theor. Comput. Sci.*, 116, January 2005.

[16] R. K. Shehady and D. P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, 1997.

[17] P. A. Szekely. Retrospective and challenges for model-based interface development. In *Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces (CADUI'96)*, 1996.

[18] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann, 2006.

[19] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical report, The University of Waikato, April 2006.

[20] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. *Formal methods and testing*, 2008.

[21] S. Winter. *Modellbasierte Analyse von Nutzerschnittstellen.* Dissertation, Technische Universität München, München, 2009.

[2]`http://www.bugzilla.org/docs/3.6/`