Specification based testing of automotive human machine interfaces

Holger Grandy, Sebastian Benz
{holger.grandy, sebastian.benz}@bmw-carit.de
BMW Car IT GmbH

Abstract: Model based testing promises systematic test coverage in a continuous testing process. However, in practice, model based testing struggles with informal specifications, different software variants and large applications. In this paper, we present a solution to overcome these hurdles in the area of automotive infotainment systems using domain specific languages in combination with model transformations. Our approach is to define specific languages on different abstraction levels. We start with a variant-spanning user interface specification that is structured and formal, but not yet sufficient for automated testing. We use model transformation to stepwise enrich and refine these models into more specific test models. The approach has been developed at BMW Car IT and is currently used in the development of new infotainment systems.

1 Introduction

At the moment, manual testing is the most often used mean of verification in industry. During the development of a system, 50 percent of the time and more than 50 percent of the total cost are expended for testing [MSBT04]. In contrast to manual testing, model based test case generation is a promising approach to reduce the effort of test case creation and enables the systematic selection of test cases. Furthermore in combination with an automated execution of test cases the automatic generation of test cases allows a continuous testing process during the development of a system. Especially for systems with different configurations and region specific variants, like an automotive infotainment system, the automatic generation of test cases is reasonable, because each variant requires its own test suite.

There exists exhaustive research on model-based test case generation, e.g. [BJK+05, Utt07] give a good overview. The researched topics range from modeling notations, test generation algorithms, and test selection criteria. However, in practice, model-based testing is still not widely spread. The reasons why model-based testing is not applied are often profane. Specifications are informal and therefore prohibit the generation of test cases. There are no adequate tools that support modeling and test case generation. Available modeling languages are often too general in their abstraction level and thereby lack the required conciseness to be able to maintain large system models.

The contribution of this paper is an overview on how these problems can be solved using the combination of domain-specific languages and model transformations. We present

our approach of specification based test case generation for infotainment systems that we successfully integrated into the development process at BMW Car IT.

The paper is structured as follows: Section 2 introduces the challenges of test case generation for infotainment systems. Section 3 presents our approach for model-based test case generation and Section 4 gives an outlook on future work and concludes.

2 Description of the task

In the area of model-based testing, research and academia focus on appropriate formal modeling notations, on test generation algorithms, and on model coverage criteria. However, these are currently in practice of lesser concern. The main task is that currently in practice one has to deal with sometimes incomplete and constantly evolving specifications that often do not have the required degree of formalization to generate test cases. It is not realistic to believe that in the near future full-blown exact and correct formal logical models for every software component e.g. in a current premium-class vehicle will exist. One has to deal with these issues first in order to establish model-based test case generation in practice. Our experience showed, that there are further reasons why model-based test case generation is often not used in practice:

- Specifications are hand-written and therefore often incomplete, inconsistent, supplemented with informal information and error prone.
- Specification and implementation differ most of the time due to constant evolving on both sides and due to different development cycles.
- More formal notations used for specification and test case generation (e.g. Temporal Logics and Model Checking like [ZML07]) cannot be easily applied for the specification of a large infotainment system, because of
  - steep learning curve and experience in formal methods required
  - no tool support for collaboration, e.g. change requests, approval processes or version management
  - insufficient tooling support to handle large models

Furthermore, there are intrinsic challenges in the automotive infotainment domain. First of all, user interfaces of automotive infotainment systems are large. For example, the user interface of the current BMW 7-Series consists of more than 4000 different UI elements (screens, lists, buttons, popup messages, transitions, conditions, texts, ...). Thus, the effort of handling such a large specification is high in order to keep the specification consistent and correct.

Another main challenge is to handle the large space of different UI variants. The same user interface is integrated into different car variants and product families. This results from different variants for the same car (“variant”) - e.g. whether or not a navigation system is

equipped - and from variants caused by different software product lines ("product line") in different car families within the BMW Group.

Product lines may share common functionality - for example, the navigation system. However, the user interfaces of different product lines differ in their look and feel. For example, the main menu behavior in the user interface of the BMW vehicles is a plain list. In other product lines it could be possible to have a cyclic structure, which rotates as the user operates the IDrive controller. In such an UI it would be possible to jump from the last element to the first one in contrast to the current BMW UI.

The functional parts of the UIs are specified in a product line spanning manner. Different product lines and variants are encoded by parameters, for example, one parameter describes whether there is a navigation system integrated. The user interface of the current BMW 7-series is described by 1000 such parameters.

However, test case generation requires fully executable models. These models must integrate the common functional specification as well as the product line specific specifications of the look and feel. Furthermore, they must be adaptable in order to overcome temporary differences between specification and implementation. For example, when in the specification the order of the main menu buttons "radio" and "navigation" is changed: If this change has not yet been implemented and a tester generates test cases, all tests will fail that use these buttons. However, if a tester wants to test whether all texts in the radio menu are shown correctly, the generated test cases cannot be used. Thus, even if the tested system properties are independent of the button ordering, the test case generator cannot be used. The tester must be able to revert this change on short notice in order to be able to test the radio sub menu.

3 Approach of specification based testing

Our approach is to separate product line specific behavior models and product line independent behavior models. We define domain-specific languages for each of these behavior models. The advantage of DSLs in comparison to general purpose languages, such as UML, is that these are directly tailored to the problem domain. Furthermore, DSLs enable the integration of OEM specific elements into the language. This results in concise models that describe only the relevant parts. We use DSLs at different levels:

- **Dialog Model** A product line independent dialog model that is used to specify common functionality and structure. The model consists of two parts, the abstract dialog model that describes the menu tree and an application model that describes the interfaces of the underlying applications.
- **Delta Model** A delta model enables the temporary adaption of the specification:
  - describe differences between specification and implementation
  - enrich the specification with missing information
  - replace informal parts with formal ones

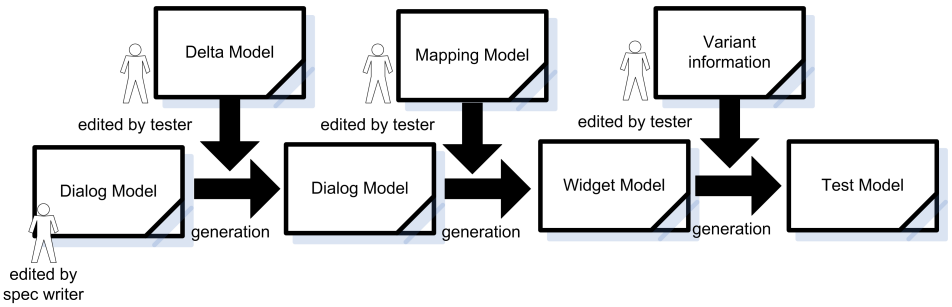

Figure 1: A workflow for specification based testing

- **Widget Model** The widget model describes product line specific widget behavior, such as list or button behavior.
- **Mapping Model** The mapping model maps entities from the dialog model to their implementation in the widget model. For example, the abstract main menu in the dialog model is mapped to a BMW specific main menu widget implementation.

We use automated model transformations to create a product line and variant specific test model based on these three models. Figure 1 gives an overview over the workflow. A test model is created in three steps:

1. The dialog specification is transformed into a product line specific dialog model. and the delta model is applied to the dialog model to adapt the specification to the current state of the implementation.
2. The product line specific dialog model is transformed into an executable widget model by applying product line specific mapping rules.
3. The variant of the system under test is encoded into the widget model, resulting in the test model.

3.1 The dialog model

The dialog model describes the structure and dynamic behavior of the user interface. It consists of a structural model that describes the abstract menu tree and an application model that describes the interfaces of the underlying applications. The structure is described by abstract screens that consist of containers and elements. Containers are, for example, lists and elements are, for example, buttons. Changes between different screens are described using transitions which can have additional actions or guards. As an example, we will use the above mentioned example of the cyclic or plain main menu. Listing 1 shows the specification of the main menu dialog structure.

```
module modulemain {
  screen mainscreen productLine=ALL {
    title=TEXT_TITLE
    container container_main_menu type=list {
      element ButtonRadio text=... comment=...
        transition target=moduleradio changerequest=...
      element ButtonNavi variant=VARIANT_NAVI ...
        transition target=modulenavi comment=...
      element ButtonBMWOnline disabledIf="NO_INTERNET_CONNECTION" ...
    }}
}
module moduleradio {
... }
```

Listing 1: Dialog model.

The application model describes the interface of applications that are part of the infotainment system. This is e.g. the playback information of tracks in a CD player. It is used to specify all data types required in the UI behavior and application specific actions. One example are events, which are triggered by the user by using the IDrive controller or by external sources like an incoming call. An application interface consists of:

- **Data models:** represent data that is required by the UI. For example, a list of CD tracks that is shown in the track list screen.
- **Actions:** can be triggered by the UI. An action can either change the data model or trigger an action in the application, for example, to start the playback of a CD.
- **Events:** represent inputs that can be triggered by the application. Examples are joystick inputs, or check control messages.

Listing 2 shows a simplified example for such an application model, describing the interface of the IDriver controller and the CD player.

```
application IDrive{
  event IDriveEvent {
    String id}
  IDriveEvent left = new IDriveEvent("left");
  IDriveEvent right = new IDriveEvent("right");
  boolean VARIANT_NAVI = true
  boolean VARIANT_HEADUP_DISPLAY = true}

application CDPlayer{
  int currentTrack = -1
  list(String) tracks = {'Track 1', 'Track 2'}
  action playTrack(int trackIndex){
    currentTrack = trackIndex}}
```

Listing 2: Application model.

The application model further contains all the variables representing the configuration of the variant of the system under test. For example the information, whether or not a navigation system is built into the system under test is represented by an boolean variable VARIANT_NAVI. This variable is used to specify the variant of the button ButtonNavi in Listing 1.

3.2 Delta Model

The goal of the delta model is to provide an easy way to describe changes in the dialog model, when changes in the original specification are not quickly possible because of e.g. development process related issues. It is also a good way to document change requests. The goal is not to provide an additional specification. Instead, the goal is to give testers a tool that they can use to perform quick changes in the specification, like e.g. changing the order of elements or changing conditions:

```
move element 'ButtonRadio' to position 3
move element 'ButtonNavi' to position 2
add disabledIf 'NO_RADIO_RECEPTION' to element ButtonRadio
```

Listing 3: Delta model.

3.3 Widget Model

The second step in our approach is the generation of product line specific test models. For test case generation we use state machines as abstraction which corresponds to existing approaches, e.g. [AO99, EKRV06, MAMS06]. The advantage of state machines is that there exists a wide range of techniques for test case generation. However, the problem is that state machines or statecharts [Har87] are not the right abstraction to describe a GUI. Modeling an automotive user interface using state machines is tedious because expressing GUI concepts often results in large and unmaintainable models. Our goal is to keep the statechart interface (the notion of events, transitions, active states, guards, ...) in order to generate test cases. The solution is to integrate domain knowledge into the state machine metamodel. One can think of the widget model as domain specific state charts.

In this language, we are defining product line specific statechart states, which already implement certain functionality of the product line. For example, a list typed UI element in the specification is mapped to a statechart combined state which additionally already implements focus handling in the list per default. For a product line with a cyclic main menu, we map to a specific state that implements cyclic focus handling. For a BMW product line, we map to a state that implements a plain list focus handling strategy. This is done for example by adding transitions to the statechart state for the list that dynamically change their targets according to the currently selected menu element. Such a behavior is not possible in standard statechart semantics.

Figure 2 shows the differences in more detail. The right hand side of the figure shows our domain specific statechart models. The specific state MainMenuWidget implements the specific behaviour of the main menu. The extended behavior of the state is executed when an event is handled by this state. The two routines shiftright and shiftleft change the targets of the two transitions for the events left and right. For example, if the current state is Navi (this is the current situation in the figure) and the right event is fired, the active state becomes Settings, the target for transition with the left event becomes Navi and so on.

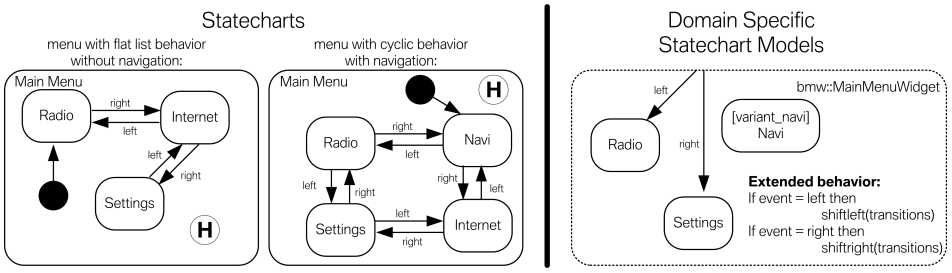

Figure 2: standard statecharts vs. domain specific statecharts

Furthermore, the widgets can be guarded (e.g. state Navi in the figure). The shifting routines will select their next transition target according to the value of that guard.

3.4 Widget Mapping Model

The widget mapping model is responsible for bridging the gap between the dialog model and the domain specific statechart states in the widget model. It contains mapping rules, which define the relation between the two. Listing 4 shows as an example the definition of mapping rules, which map every screen in the dialog model to a BMW product line specific state with the name ScreenWidget. An exception is the state with the name Nav_MapView which implements the view for the navigation map. This screen has a special behavior (for example it shows a toolbar at the left side for controlling the map). So this screen is mapped to a special state of type MapViewScreenWidget. The same is true for the above mentioned main menu example. Every list in the dialog model is mapped to a ListWidget, except the main menu, which is mapped to a MainMenuWidget.

```
screen to bmw::ScreenWidget except {
  name Nav_MapView to bmw::MapViewScreenWidget}
list to bmw::ListWidget except {
  name List_Main_Menu to bmw::MainMenuWidget}
```

Listing 4: Widget mapping model.

4 Summary

The presented approach enabled us to introduce model based testing in our UI development. We demonstrated that model transformations and automatic code generation, paired with extendable textual models for e.g. application specific parts are suitable to enable testers to test certain aspects in more detail. This also bridges the abstraction gap between test models and specifications. Specifications benefit from our approach as well, because a transformation to executable models enables simulation and validation, too.

We made heavy use of the eclipse modeling framework (EMF)[1] and Xtext[2] to define the meta models and textual syntax of our languages. Xtext has the advantage that it supports generation of editors with highlighting, code completion and syntax checking. Our experience is that especially textual modeling languages are a good way to describe large models, where graphical editors become unusable. The main advantage is that for textual languages already a large set of tools exists to support distributed development.

Functional interaction between software components, which is generally error prone, has significantly increased in the past and will increase further. We already did work on special methods testing of interaction scenarios [Ben07]. For the future, we are planning to use the results of that work in conjunction with the models of this paper.

Acknowledgments: We thank Michael Fitzner, Markus Hillebrand, Maximillian Leinweber and the UI specification and test team at BMW for their work in this context. We also thank Harald Heinecke and Jürgen Steurer for their helpful comments.

References

[AO99] Aynur Abdurazik and Jeff Offutt. Generating Test Cases from UML Specifications. Technical Report ISE-TR-99-09, George Mason University, 1999.

[Ben07] Sebastian Benz. Combining test case generation for component and integration testing. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 23–33, New York, NY, USA, 2007. ACM Press.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[EKRV06] Juhan P. Ernits, Andres Kull, Kullo Raiend, and Jüri Vain. Generating Tests from EFSM Models Using Guided Model Checking and Iterated Search Refinement. In *Formal Approaches to Software Testing and Runtime Verification*, pages 85–99. 2006.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[MAMS06] P. V.R. Murthy, P. C. Anitha, M. Mahesh, and Rajesh Subramanyan. Test ready UML statechart models. In *SCESM '06: Scenarios and state machines: models, algorithms, and tools*, pages 75–82, New York, NY, USA, 2006. ACM.

[MSBT04] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing, Second Edition*. John Wiley and Sons, New Jersey, June 2004.

[Utt07] Mark Utting. *Practical Model-Based Testing*. Morgan Kaufmann Publishers, San Francisco, 2007.

[ZML07] Hongwei Zeng, Huaikou Miao, and Jing Liu. Specification-based Test Generation and Optimization Using Model Checking. In *TASE '07: Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 349–355, Washington, DC, USA, 2007. IEEE Computer Society.

[1] http://www.eclipse.org/emf
[2] http://wiki.eclipse.org/Xtext