Model-Based Testing for the Menu Behavior of Automotive Infotainment System HMIs

Linshu Duan
Ludwig-Maximilians-Universität München
and
AUDI AG
linshu.duan@audi.de

Heinrich Hussmann
Institut für Informatik
Ludwig-Maximilians-Universität
München
heinrich.hussmann@ifi.lmu.de

Dieter Niederkorn
and
Alexander Höfer
Infotainment System Testing
AUDI AG
dieter.niederkorn@audi.de
alexander.hoefer@audi.de

ABSTRACT

Testing the graphical human machine interface (HMI) of automotive infotainment systems has shown to be costly and challenging due to its large function scope, high complexity and multiple variants. To ensure the quality and reduce testing costs we are working on a model-based testing concept for graphical HMIs of infotainment systems. In our work the short form "HMI" is used for the term "graphical HMI". In this paper, we present some preliminary results of our model-based testing research. We firstly introduce the classification and distribution of HMI errors. This statistic shows that errors in the menu flow construct an essential part of HMI errors. In this paper we focus on the detection of this kind of errors. For this UML state machine has to be extended to describe the menu behavior, so that valid tests can be generated from its instances. Common coverage criteria for the state machine can not produce efficient tests for infotainment system HMIs. Therefore, we discuss some defined adequacy criteria for infotainment system HMI tests. At last we briefly introduce how we use software product-line approaches to integrate variability into the model-based HMI testing concept.

Author Keywords

Model-Based Testing, Test Models, HMI-Testing, Software Product-Line, Statechart with Variability

INTRODUCTION

Infotainment system HMIs of new generations have a very wide function scope and can contain more than 2000 menus and 100 pop-up menus. They usually have many variants caused by different markets, product-lines, individually configurable features and equipments.

To reduce the testing costs and ensure a systematic code coverage, we are working on a model-based and automated testing concept specific for infotainment system HMIs. The concept has been introduced in previous papers [5] [6]. In this paper we present some new results of our ongoing work.

This paper is organized as follows. In order to clarify which kinds of HMI errors can occur in practice, we have evaluated some parts of a past HMI development project. We will present our results of the error classification and a rough distribution.

UML statechart, which graphically represents a state machine, is widely used for the development and specification of infotainment system HMIs. However, the standard UML state machine from the OMG (Object Management Group) [14] is not sufficient for specifying the menu behavior so that valid and automatable tests can be generated from the specification. We firstly introduce the test-oriented HMI specification in which the menu behavior model is located and then required extensions of the state machine for creating a testing-ready menu behavior model.

Test generation based on common coverage criteria can not produce adequate tests for infotainment system HMIs. We firstly introduce the generated tests based on two chosen common coverage criteria and then explain the specific adequacy criteria for infotainment system HMI tests.

Finally, we will introduce the variability of infotainment system HMIs and how variability handling can be integrated into our model-based testing concept.

RELATED WORK

A number of research efforts have addressed the model-based testing of GUI applications [17], [16], [1], [10] and [11].

The NModel framework introduced in [2] supports finite state machine (FSM) models and automatic test generation for GUI-driven applications. In this approach binding user data or data exchange with external components are not considered.

In [17] a model-based software testing method for web applications is presented. This method focuses on testing the functionalities of the front end of web applications, i.e., the linking behavior of the links and forms, which is modeled with statecharts. However, this method has not yet found solutions to the problems of modeling the back-ends of a web application. Without the behavior of the back-end, a gener-

ated test only describes a possible sequence contained in the model and does not consider the conditions. So generated tests are usually infeasible for the test automation. For infotainment systems HMIs the behavior of the back-end is very complex and error-prone. Testing this logic automatically is a very important test purpose.

The concept described by Memon [11] does not separate the menu flow behavior and the physical structure of the HMI, which is inappropriate for specifying the infotainment system HMI. Different variants of infotainment system HMIs are usually developed in one model as one product family. The variants in the family usually have the same or very similar behavior but different physical HMI elements or structure. Separation of them is an important requirement in the HMI specification.

In [10] LTS (labeled transition system) with action-word and key-word technique is used. The concept separates the specification of the business logic and the presentation logic. The action model contains action-words, which are abstract events. They describe the behavior of the system. The refinement model contains both action-words and key-words. Key-words are user events or menu navigation, which are performed by concrete HMI elements. Refinement model describes also how action-words can be refined with key-words.

In [1] the authors extend the state machine with regular expressions to consider not only correct HMI actions but also incorrect transitions.

In [12] and [16] some specific coverage criteria for HMI testing are introduced in addition to common coverage criteria [13] [7].

Works [18] and [9] focus on variability of SW products with product-line approaches in the domain of SW development. In [9] a state machine contains all potential features of all products in the product family. The goal is to choose required sub states from the state machine, resolve the relations and generate code for a certain product. In [18] for each feature of the product family there is a state machine available. The task is to select the required state machines, find the connections of them and generate code for a certain product.

ERROR CLASSIFICATION AND DISTRIBUTION

Past infotainment system projects containing advanced and complex HMIs are chosen for a statistical analysis. HMI error tickets are evaluated which were created during the development phase. Figure 1 presents the classification and distribution of the errors.

About a quarter of HMI errors are in the menu logic, so-called menu behavior errors. They appear in the form of switching to an unexpected menu in response to some inputs from the user or underlying applications. In practice of automotive HMI domain, the menu behavior is usually specified with statechart models.

More than the half of the HMI errors are in the views and contained graphical elements. Views are usually called screens in the automotive domain.

A view usually contains static contents such as a title and

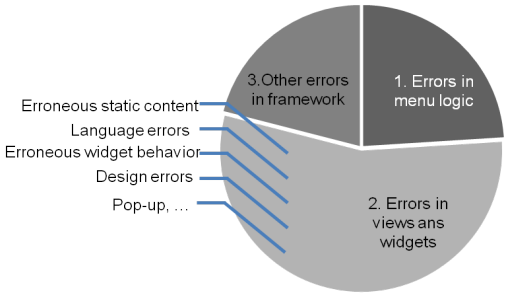

Figure 1. Error classification and distribution

subtitle as in Figure 2, which are displayed at any time and in any context. The error statistic has shown that an essential part of HMI errors arise from erroneous static content such as a missing text. They can be easily found, if menu behavior tests are extended with sub tests verifying the static contents. Required information for testing static contents are specified in the presentation layer of the test-oriented HMI specification, which will be introduced in the next section. Simple image processing methods can be used to get the presented texts from the display [5].

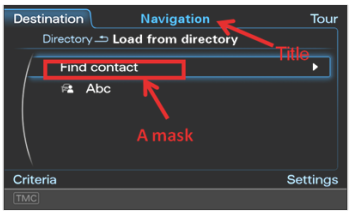

Figure 2. Static content of a menu and a mask

Infotainment system HMIs are usually available for many languages. Language errors are either contained in the localization database or caused by erroneous linking between entries in the localization database and representing widgets. Errors caused by erroneous linking and representing widgets can also be found by static content tests.

Infotainment system HMIs contain usually a lot of advanced widgets with dynamic behavior. Dynamic widgets lead to a lot of HMI errors. The widget behavior can also be defined with statechart models. Widget behavior tests can be generated from these models and extended to menu behavior tests. However our prototype of a widget behavior model has shown that modeling widgets can be very work-intensive and time-consuming. It only makes sense to model especially errors-prone widgets and test them automatically. Therefore it is very important that the menu behavior and the widget behavior are separated in different models in a HMI testing concept.

There are many other errors, which are not in the HMI but directly affect the HMI behavior. They are either due to the HMI framework or underlying applications. For example, the phone application has sent an empty string as a contact name to the HMI or the switching between different

menus have some delay because the bus system is heavily loaded. Modeling the behavior of underlying applications or the whole infotainment system is infeasible for infotainment system HMIs. This category of errors can not be found with the concept.

Preliminary evaluation results provide only a picture of the error classification and contribution. One can define the categories in a very different way and we believe, statistic of other projects can deviate from current results.

TEST-ORIENTED HMI SPECIFICATION

In the last section, we have introduced that menu behavior errors construct an essential part of HMI errors. To detect menu behavior errors, a test model has to be available which specifies the expected menu behavior. In our concept, such a test model is called menu behavior model and is arranged in the behavior layer of the test-oriented HMI specification.

A test-oriented specification [5] [6] is a HMI specification, which contains sufficient information for testing purposes. It's constructed with a layered structure as shown in Figure 3:

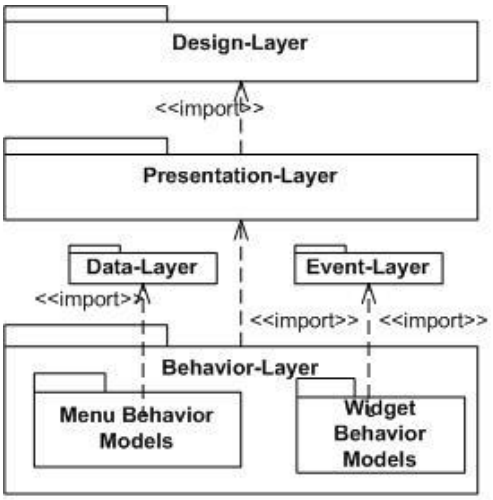

Figure 3. Layers of the test-oriented specification

The presentation layer contains testing-relevant information about screens and their graphical elements thus: potential events which can be triggered by a screen and the abstract content and structure of a screen. Data and event layers contains variables and events used in other layers. The widget behavior models describe the behavior of complex widgets. The design layer which contains design information is optional. It is only required if design tests should be performed. This is not the focus of our work. As shown in Figure 3, menu behavior models are separate to other models or information. This separation provides the possibility to specify the menu behavior and perform menu behavior tests independently.

THE MENU BEHAVIOR MODEL

In this section we focus on the specification of the menu behavior and introduce some required extensions of the UML state machine for specifying the menu behavior with testing purposes. We introduce our extensions based on the state machine definition from the OMG [14].

ViewState

Currently three kinds of states are distinguished in the state machine: simple state, composite state and submachine state. A new kind of state: ViewState has to be extended for describing the HMI menu behavior. A view state is a special kind of simple state signifying that the current state is associated with an abstract screen in the presentation layer. A views state has an attribute of type string, which is the name of the associated view in the presentation layer. When a view state is active, the associated view has to be displayed.

PreStepsCondition

In many situations some user actions are only enabled, if some other actions are previously performed. For example, entering a city name as navigation destination is only enabled, if a country name has been entered. A test which enters a city name and starts the guidance without to enter a country name before is an invalid test. PreStepsCondition is extended into the state machine, which indicates this dependency for the test generator. Transitions which need other transitions as previous steps, have to be labeled with a PreStepsCondition. Transitions fulfilling some PreStepsConditions have to be labeled with actions making these conditions true. A PreStepsCondition contains a function, which evaluates to a boolean value. For each PreStepsCondition, there must be at least one transition labeled with an action which makes the contained function true.

RuntimeCondition

An infotainment system HMI is not a closed application such as a simple calender, which contains the complete behavior logic in itself. During the runtime, an infotainment HMI communicates with the underlying applications almost all the time. The menu behavior is strongly dependent on the runtime data. However at the time of creating models and generating tests, the runtime data and consequently the completion of conditions are unknown. For example, when a user has entered the destination completely and started the route guidance, the screen with the map and calculated results should only be shown after that the calculation is finished. However, the calculation is performed by the underlying application. That means, the transition pointing the view state associated with the map screen is only active if a condition is fulfilled during the runtime by the underlying application. A new type of condition "RuntimeCondition" has to be extended to describe this dependency. A RuntimeCondition contains a function, which evaluates to a boolean value. The function has to be bound with runtime variables, from e.g. the interface between the head unit and the underlying application. The function can only be fulfilled by the runtime variables.

Some new event types

Figure 4 show the event types defined by the OMG for the UML state machine.

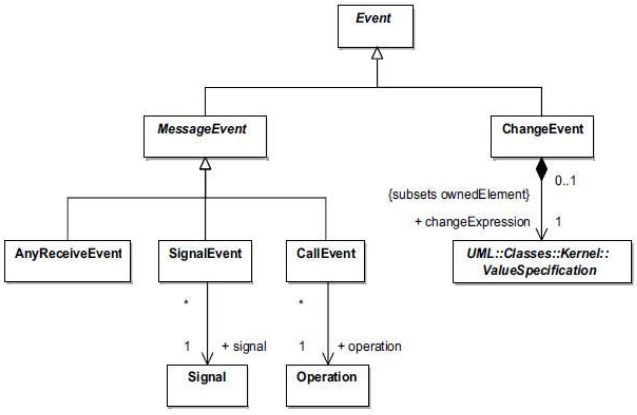

Figure 4. Event types in a UML state machine

We have defined different types of events depending on their sources. For instance, the type ApplicationEvent is defined for events initialized by underlying applications. An application event can be a message event or a change event. In events which are performed by users we distinguish two types: GlobalEvent and ReacionEvent. Global events can be performed via the control unit anytime and they are effective for any HMI states. For example, for switching between different infotainment system functions such as radio and navigation, buttons are provided in the control unit, which trigger global events. In contrast to global events, a reaction event can only be triggered in certain screens. Distinguishing different event types is very important for the test generation and instantiation.

A lot of functions of infotainment systems require user input-data, e.g. a phone number to dial or a destination for the guidance. Representatives for each user input-data equivalent class have to be tested. User input-data for tests should better be separately defined independent from the test model (not a part of the test model). In this way, to change the user input-data for different testing purposes or phases would not lead to some model changes. On the other hand, the separately defined user input-data can be reused for other tests.

To bind the user input-data, "UserInputEvent" has to be extended into the state machine. Currently we have defined equivalent classes for correct user inputs and unexpected user inputs.

COVERAGE CRITERIA FOR INFOTAINMENT SYSTEM HMI TESTS

We have implemented test generation algorithms based on some common coverage criteria in order to evaluate their adequacy for infotainment system menu behavior testing.

As explained, the menu behavior of an infotainment system HMI is strongly dependent on the runtime data. For an infotainment system HMI with 1000 menus, up to 250 conditions states are needed to model the dependency of the menu behavior on runtime data. So we have implemented a generation algorithm based on the branch coverage, which means all outgoing transitions of existing condition states have be tested. Our implementation is based on the depth search and allows currently each cycle for once. The generation results have shown that the generated tests can cover all branches, the number of generated tests is limited and the tests are very short. Generated tests are very unusual user scenarios.

Infotainment systems are very function-oriented. Each function e.g. starting the route guidance is usually accessible on one unique menu. We have implemented an algorithm, which generates all paths to a destination menu in which a certain function is accessible. All-path coverage could produce infinite tests. To limit the number of generated tests we allow each cycle only once. The generated tests cover all possible paths to the destination menu. However, most of the tests are in the same equivalent class, which means, the error could already be found with only one of the tests. Execution of all tests is unnecessary and impossible in the testing life due to very limited testing time and resources. Evaluation of other common coverage criteria e.g. transition coverage and HMI-special criteria as introduced in [12] and [16] is planned.

We firstly discuss criteria of adequate and efficient tests for infotainment system HMIs.

One of the most important requirements in premium HMIs is a faultless textual and graphical representation. So viewing all existing menus for all languages at least once would be the first criterion for infotainment HMI tests. We could derive the ViewState-coverage from this criteria.

Reusability of menus is very common in the implementation of infotainment system HMIs. For example a menu representing the contacts exists only once and is accessible from both the navigation and address book context. The menu shows different color and widgets depending on the accessing context. The reuse of menus could be very error-prone. So tests accessing reused menus from different ways can be very efficient to find errors.

We are still working on the definition of infotainment system HMI-specific coverage criteria based on the found adequacy criteria.
At last we would like to show a small example of a generated test, which firstly enters a destination and then starts the route guidance. We use the syntax [] for an expected menu name and () for a test step:
[navMain]−> (enter on widget "country") −> (wait(5ms)) −> [navCountryList]−> (userInput_selectCountry_correct) −> (enter) −> [navMain]−> (enter on widget "startRG") −> [navRgStarted]

INTEGRATING VARIABILITY INTO MODEL-BASED HMI-TESTING

A software product-line (PL) [3] [8], also called system family, is a set of software systems sharing common features that satisfy the specific need of a particular market segment and that are developed from a common set of core assets in a prescribed way.

Infotainment system HMIs are multi-variant products. The variability results from product series such as different generations, market variants such as for Europe or Asia, configuration variants such as with or without DVD player and system variants such as standard resolution with normal display or higher resolution with larger display. In practice many of these variants are developed in one project due to a large set of commonalities in features, looks and behaviors. That means, an HMI model in such a project describes all features, looks and behaviors potentially required for different variants e.g. for both Europa- and Asia-market. A product, which is created from such a model exactly satisfies one variant e.g., a standard system with navigation feature for the Europa market. For these reason, the model-based HMI testing concept has to be extended to support the variability.

We reuse feature models to extend the test-oriented HMI specification for the variability management. Feature models (FMs) allow us to describe both commonalities and differences of all products of a PL and to describe the relationships between them. A FM configuration (FMConf) is an instance of a FM that describes the properties and functionality of a product. In [4] FM and FMConf are described in details. We extend the in [4] defined FM and FMConf with distinction of two kinds of children: functional features and non-functional features. A functional feature can be e.g. the feature radio or navigation. A non-functional feature can be a variant feature, e.g. Europa variant or Asia variant. Usually, the relationship between functional features is or-relation and the relationship between variant features is alternative-relation. Figure 5 shows a strongly simplified example: f stands for a functional feature and v stands for variant feature.

To extend menu behavior models for variabilities, some

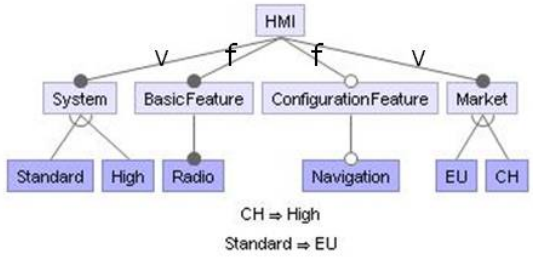

Figure 5. A FM with functional and non-functional features

new elements have been introduced as extensions for UML state machines. A feature composite state is a special kind of composite state in which the behavior of a function feature is described. A feature composite state is always related to a functional feature in the FM. An HMI allows inter-feature activities e.g., from the feature navigation the user can switch to the feature telephone and choose an address of a contact as destination. Therefore, in the menu behavior model there can be transitions between two feature composite states, which are called inter-feature transitions. Furthermore, variation points and junction points which are originally defined in [15] for activity diagrams are extended for the state machine. Each variation point is related to a child node which is marked with v in the FM e.g. "system" in Figure 5. Each outgoing transition is related to one of the contained variant features e.g. "standard" or "high". A variation point can only be used within a pair with a junction point, which merges the distinction of the variant feature behaviors.

Also the presentation layer has to be extended for variabilities. Parameterized inheritances are used in the abstract description both of the screen structure and events which can be potentially triggered by the screen. Since presentation layer is not the focus of this paper, it is thus not further discussed.

Each infotainment system test bench conforms to a valid FMConf. For instance, a test bench is a "high" system for Europa with the basic feature radio and configurable feature navigation as shown in Figure 5. In a testing farm test benches for many configurations are available. Since many configurations share a common set of functional or non-functional features, avoiding the redundancy is one of the most important requirements for the test generation and test execution. Therefore the test generation is composed of two steps. Firstly, algorithms traverse the whole test model and generate partial tests for all required functional and non-functional features. Then tests are created from these partial tests for all required configurations. In this way redundant generation of common features are avoided. If changes are only carried out in a sub set of the features, the test generation is able to regenerate partial tests from the affected partial test model and the second step has to be executed for the affected features. Avoiding redundant test execution is especially important in industrial practice due to limited test resources.

CONCLUSION

In this paper, we introduced some preliminary results of our model-based testing research for infotainment system HMIs. Error statistic and some additional elements needed for modeling the menu behavior were introduced. We also discussed which tests are adequate and efficient to detect errors in our area. Furthermore, we have briefly introduced the main ideas how we extend the model-based HMI testing for variabilities.

ADDITIONAL AUTHORS

REFERENCES

1. F. Belli. Finite-state testing and analysis of graphical user interfaces. In *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering*, page 34, Washington, DC, USA, 2001. IEEE Computer Society.
2. V. Chinnapongse, I. Lee, O. Sokolsky, S. Wang, and P. L. Jones. Model-based testing of gui-driven applications. In *Software Technologies for Embedded and Ubiquitous Systems*, volume 5860/2009, pages 203–214. Springer-verlag New York Inc, 2009. 7th IFIP WG 10.2 International Workshop, SEUS 2009 Newport Beach, CA, USA, 2009 Proceedings.
3. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in SE. Addison-Wesley, 2002.

4. K. Czarnecki. Generative programming: Methods, techniques, and applications. In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*, ICSR-7, pages 351–352, London, UK, UK, 2002. Springer-Verlag.

5. L. Duan, A. Hoefer, and H. Hussmann. Model-based testing of automotive hmis based on a test-oriented hmi specification model. In *Proceedings of the the 2nd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2010), August 22-27 2010, Nice, France*. IEEE, Aug. 2010.

6. L. Duan, H. Hussmann, and A. Höfer. A test-oriented hmi specification model for model-based testing of automotive human-machine interfaces. In *GI Jahrestagung (2)*, pages 339–344, 2010.

7. C. Gaston and D. Seifert. Evaluating coverage based testing. In *Model-Based Testing of Reactive Systems*. Springer-Verlag New York, LLC, 2005.

8. H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. The Addison-Wesley Object Technology Series. Addison-Wesley, 2004.

9. A. Gonzalez and C. Luna. Behavior specification of product lines via feature models and uml statecharts with variabilities. In *2008 International Conference of the Chilean Computer Science Society*, pages 32 – 41. IEEE Computer Society, 2008.

10. A. Kervinen, M. Maunumaa, T. Pkknen, and M. Katara. Model-based testing through a gui. In *In Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005), number 3997 in Lecture Notes in Computer Science*, pages 16–31. Springer, 2006.

11. M. A. M. An event-flow model of gui-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, 2007.

12. A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for GUI testing. pages 256–267, 2001.

13. J. Offutt and A. Abdurazik. Generating tests from uml specifications. page 76, 1999.

14. OMG. Omg uml, superstructure. `http://www.omg.org/spec/UML/`.

15. S. Reis and K. Pohl. Wiederverwendung von integrationstestfällen in der software-produktlinienentwicklung. *Inform., Forsch. Entwickl.*, 22(4):267–283, 2008.

16. H. Reza, S. Endapally, and E. Grant. A model-based approach for testing gui using hierarchical predicate transition nets. In *Proceedings of the International Conference on Information Technology*, ITNG '07, pages 366–370, Washington, DC, USA, 2007. IEEE Computer Society.

17. H. Reza, K. Ogaard, and A. Malge. A model based testing technique to test web applications using statecharts. In *ITNG '08: Proceedings of the Fifth International Conference on Information Technology: New Generations*, pages 183–188, Washington, DC, USA, 2008. IEEE Computer Society.

18. N. Szasz and P. Vilanova. Statecharts and variabilities. In P. Heymans, K. C. Kang, A. Metzger, K. Pohl, P. Heymans, K. C. Kang, A. Metzger, and K. Pohl, editors, *VaMoS*, ICB Research Report, pages 131–140, 2008.