Towards a GUI Test Model Using State Charts and Programming Code

Daniel Mauser[1], Alexander Klaus[2], and Konstantin Holl[2]

[1] Daimler AG, Ulm, Germany
[2] Fraunhofer IESE, Kaiserslautern, Germany
daniel.mauser@daimler.com,
{alexander.klaus,konstantin.holl}@iese.fraunhofer.de

Abstract. Modern human machine interfaces provide a sophisticated structure and logic to ease their use. As they are the only mean to control the system behind, extensive testing and highest quality is required in the automotive domain. A common testing approach in literature is to derive the necessary test cases from a formal model. However, redundancy and data dependency still hinder manual modeling in the industrial context. In this paper, we present preliminary work to address these obstacles. As a first step, we combined depictive state charts with reusable programming code. We modeled parts of the graphical user interface of a state-of-the-art infotainment system and successfully generated a test suite that covers our testing goal to reach each button at least once.

Keywords: automotive, human machine interface, model based testing.

1 Introduction

An automotive human machine interface (HMI) provides system functionality to the user. The main interface is usually represented by a graphical user interface (GUI). Figure 1 shows an example for such a GUI including a possible screen structure. According to [1], a GUI "is essential to customers, who must use it whenever they need to interact with the system". However, testing automotive HMIs leads to more challenges than testing standard PC applications, caused by the special characteristics of automotive HMIs, e.g., the dynamic menu behavior and the large set of variants [2]. Effective usage of an automotive HMI by the user requires an effective quality assurance process. Failures during the usage while driving may lead to a distraction of the driver.

The complexity of the specification in the automotive domain is typically handled by the definition of conditions that represent the states of connected applications and devices. They consist of internal conditions, such as the selection of an option (e.g., "ESP on/off"), and of external conditions, such as the availability of a functionality (e.g., "ESP available/unavailable"). The state of the conditions influences, e.g., the availability or visibility of menu entries. As modern automotive infotainment systems comprise hundreds of specified conditions, managing the complexity manually is not feasible. Every condition can

H. Yenigün, C. Yilmaz, and A. Ulrich (Eds.): ICTSS 2013, LNCS 8254, pp. 271–276, 2013.

Fig. 1. The SUT is structured in four vertical menus that hold the clickable elements

be set to various values; this results in an extremely large number of possible combinations. Handling the amount of conditions can be done by modeling the set of conditions and its effects within a test model. It describes the conditions and its dependencies and leads to a simplification for the test engineer. Defining rules for test case derivation enables the possibility to cover the complexity of the conditions. Hence, a test generator can automatically generate test cases with the intended test coverage. This increases the controllability for reaching every menu entry which is desired to be tested. The testing goal, to ensure that all menu entries are reachable, is important for all types of integration tests.

While developing these kinds of test models, we experienced the following two major issues: The first issue is that widget behavior often depends on data such as system conditions or inserted data. One example is the entry behavior of a menu: in the system under test, menu widgets have a property "entryStrategy" that influences, which of the the containing button elements is focused the first time the menu is entered, which again depends on what buttons are actually visible at this particular moment. Test models should support efficient means to describe this data dependency. The second major issue is redundancy of behavior. Due to the modular nature of most user interfaces, atomic elements, so called widgets (buttons, menus, ...), are reused in order to ease the specification (consistent interaction concepts) and implementation (reuse of software modules). Explicitly modeling each instance of all widget types occurring in an entire system is time consuming and error prone in development and maintenance. To reduce the probability of errors, mechanisms to facilitate consistent modeling have to be applied. In software engineering, these challenges are faced by code that is structured hierarchically and modularly. Logic is encapsulated in classes that are instantiated every time this particular logic is needed. In the context of an ongoing industrial research project we currently adopt this concept for modeling HMI behavior accordingly, for test case generation purposes.

This paper is structured as follows: after discussing related work regarding applicability in our context, we present excerpts of our ongoing work to develop a modularly structured test model. The paper concludes with a first appraisal of the approach and an outlook on the intended next steps.

2 Related Work

A good overview on model based testing (MBT) in general can be obtained in [3], whereas [4] focuses on MBT of GUIs in particular. As a testing approach should easily integrate in a software development lifecycle, one crucial point is the model as basis for test generation. In the automotive domain, UML state charts are "widely established in HMI specification and development" [2, P. 24]. Various approaches for MBT of GUIs use state based modeling ([5–9]). A widely used technique for model based GUI testing is reverse engineering, i.e., executing the application and analyzing the GUI ([5–7]). However, as these approaches rely on the availability of source code, reverse engineering is not applicable in our domain. An OEM, such as Daimler, usually assigns the task of creating an HMI to suppliers and receives a package containing both hard- and software. In such cases, it is not feasible for OEMs to extract the software or to use reverse engineering, since there is no operating system supporting these techniques and the input mechanisms are different.

An approach that presumes manual modeling has been presented in [8]. The authors use domain specific state machines and model transformations to obtain product line and variant specific test models. Their solution is "to integrate domain knowledge into the state machine metamodel" [8], working with a combination of EMF[1] and Xtext[2]. Although the presented approach appears to be promising, no test case generator is available to make use of the results. Therefore, we decided to adapt the method bearing in mind available tools for generation. Established solutions are, e.g., Microsoft Spec Explorer[3] or Conformiq Qtronic[4]. Qtronic relies on hierarchical UML state machines as models, which can be enriched with a custom modeling language, which is a superset of Java [10]. Test cases are generated using symbolic execution [10]. Spec Explorer uses models created with Spec#, a variant of C# [11]. The model is then explored to create test cases and capture the intended behavior [11]. The tool distinguishes between controllable and observable actions [11]. A comparison of different characteristics of both Spec Explorer and Qtronic can be found in [10].

3 Model Structure

Similar to the basic structure introduced by [8], we combine programming code and state charts to make use of the strengths of both approaches: state charts provide a clear structure of screens and their relationships and therefore ease retracing the generated test cases. Object oriented programming code is easy to reuse and provides efficient means to specify behavior. The established product "Conformiq Designer" supports these kinds of models.

[1] www.eclipse.org/emf/
[2] www.eclipse.org/Xtext/
[3] http://research.microsoft.com/en-us/projects/specexplorer/
[4] http://www.conformiq.com/

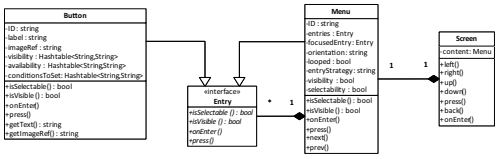

Fig. 2. The Screen and the Entry elements Menu and Button are the essential classes

We developed a "widget toolbox" containing the basic elements the user interface consists of: classes for buttons, menus and screens (see Figure 2). The **Entry** is the basic widget type that summarizes all methods that are necessary for the interaction concepts. Deriving classes have to implement methods that state on demand whether they are visible (displayed on screen) or available (visible and selectable). Entries further have to implement methods that are triggered once they are focused and once they are pressed. The **Button** represents entries that are clickable and contain content. This content can be textual (attribute: *label*) and/or a reference to an icon or symbol (attribute: *imageRef*). The attributes *visibility* and *availability* describe references to the system conditions the visibility/availability of the button object depends on. **Menus** are container elements for entries (attribute: *entries*). As menus can again contain menu objects, this class also implements the "Entry" interface and provides the respective methods. *isSelectable()* and *isVisible()* return the respective attributes that are set via constructor. Further, menus determine what containing Entry element is focused on next() and prev(). The **Screen** class is called from within the state machine. Therefore, methods for all user interactions that directly affect the content on the screen are provided. In this setup, this includes left, right, up, down, press, and back. Screen objects are the root element for all objects in the respective state space. E.g., for each application, such as Audio or Navi, a separate screen object is instantiated. As illustrated in Figure 1, the standard screen class provides three menu lines with horizontal orientation (Application Line, Playfield, Subfunction Line), which again are contained in the menu object *content*. The screen passes the user events through to the respective menus.

The **state chart** functions as main application. It consists of programming code that constructs the state machine object and a hierarchical graphical chart. Within the constructor, all necessary button, menu and screen objects are instantiated. These objects can then be used within the graphical part. An excerpt of the chart is shown in Figure 3. For modeling, the basic state chart elements are available. In our approach, we additionally distinguish between view states that refer to screen objects, and condition states that are used to evaluate system conditions. On the lowest level of the charts there are solely view states.

With the presented approach, we modeled parts of the audio application of the latest Mercedes Benz infotainment system (NTG4.5 High Edition) that provides the functionality to play music that is stored on connected media as well as to listen to radio. We assumed a fully equipped setup, including all available media types (HDD, audio and video DVD, AUX, etc.) and radio wavebands

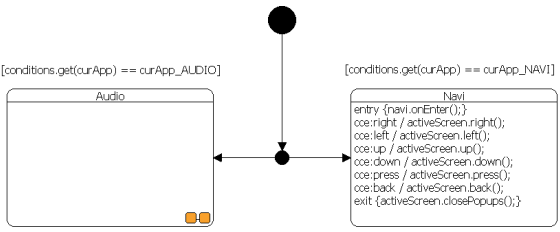

Fig. 3. The "Action" keyword of State Chart elements might refer to coded objects

(FM, LW, MW, DAB, etc.). We instantiated 15 screen, 57 menu and 138 button objects. As stated in Section 1, the testing goal is to set the system conditions to focus each button object at least once. To guide the **test case generation**, we use checkpoints in code and in the state charts. Hence, we added a checkpoint to the onEnter() method of the Button class to ensure that this method will be executed in every Button instance. The generator provided functionality to determine the necessary user interactions. To cover this state space, we had to declare 38 conditions. 26 test cases with a total of 616 test steps have been generated to fulfill the testing goal.

4 Discussion and Conclusion

In this paper, we present ongoing work on model-based black-box testing of graphical user interfaces in the domain of in-vehicle infotainment systems. We discussed basic challenges of manual model development and maintenance and pointed out that model complexity is originated in (a) the dependency of behavior on data and (b) the redundancy of logic. To address those complexity drivers we stress the need for modularly structured test models. Due to the context, manual model development is required. We developed a model structure that combines the strength of state charts to depict macro behavior with object oriented programming to allow complex data processing and modular reusability.

We developed a model to cover the audio application of a state-of-the-art infotainment system at Mercedes Benz. The disadvantageous redundancy could be handled successfully: general interaction concepts, such as the focus based navigation or list based screen structures, are described and maintained centrally; manual modeling was easy and allowed a satisfiable progress. The approach was stable regarding changes and allowed model-wide adaption of selective behavior. Due to the inheritance of object-oriented structures, several abstraction levels could be utilized: general Entries had been specialized to clickable Buttons, which again, if necessary for testing purposes, could be basis for more complex structures, such as Radio Buttons or Checkboxes. By doing so, details could be reduced to a level that could be handled manually. We successfully generated test cases using the proprietary generator "Conformiq Designer". As testing goal, we chose to focus every Button object at least once. The state charts were useful to retrace the generated test cases.

According to our experiences, the mixed approach appears to be appropriate: describing the entire behavior graphically would not have been feasible. An exclusively coded model would have been confusing and thus as error-prone as the real implementation. In future, we plan to advance the approach to automate the testing process. Next step will be to use the generated test suite as input for automatic test execution. Special attention will be drawn to the verification aspect. Due to the model structure, a data based description of all content to be displayed on screen could be exported at any time. We plan to embed this information in the test case description to provide an extensive basis to assess the system reaction, i.e., as test oracle.

References

1. Brooks, P.A., Robinson, B.P., Memon, A.M.: An initial characterization of industrial graphical user interface systems. In: Proceedings of International Conference on Software Testing Verification and Validation, ICST 2009, pp. 11–20. IEEE Computer Society (2009)
2. Duan, L.: Model-based testing of automotive hmis with consideration for product variability. Ph.D. dissertation, Ludwig-Maximilians-Universität München (2012)
3. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2007)
4. Banerjee, I., Nguyen, B., Garousi, V., Memon, A.: Graphical user interface (gui) testing: Systematic mapping and repository. Information and Software Technology (2013)
5. Morgado, I.C., Paiva, A., Faria, J.P.: Reverse engineering of graphical user interfaces. In: The Sixth International Conference on Software Engineering Advances, ICSEA 2011, pp. 293–298 (2011)
6. Arlt, S., Podelski, A., Bertolini, C., Schäf, M., Banerjee, I., Memon, A.M.: Lightweight static analysis for gui testing. In: The 23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, pp. 301–310. IEEE (2012)
7. Hackner, D.R., Memon, A.M.: Test case generator for guitar. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE Companion, pp. 959–960. ACM (2008)
8. Grandy, H., Benz, S.: Specification based testing of automotive human machine interfaces. In: Fischer, S., Maehle, E., Reischuk, R. (eds.) GI Jahrestagung. LNI, vol. 154, pp. 2720–2727. GI (2009)
9. Paiva, A.C., Tillmann, N., Faria, J.C., Vidal, R.F.: Modeling and testing hierarchical guis. In: Proceedings of the 12th International Workshop on Abstract State Machines (2005)
10. Sarma, M., Murthy, P.V.R., Jell, S., Ulrich, A.: Model-based testing in industry: a case study with two mbt tools. In: Proceedings of the 5th Workshop on Automation of Software Test, AST 2010, pp. 87–90. ACM, New York (2010)
11. 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)