Model-Based Testing of Automotive HMIs with Consideration for Product Variability

Dissertation
an der
Fakultät für Mathematik, Informatik und Statistik
Ludwig-Maximilians-Universität München

vorgelegt von

Linshu Duan

Müchen, den 28. Juni 2012

Model-Based Testing of Automotive HMIs with Consideration for Product Variability

Dissertation
submitted to
Fakultät für Mathematik, Informatik und Statistik
Ludwig-Maximilians-Universität München

by

Linshu Duan

Munich, June 28, 2012

Erstgutachter: Prof. Dr. Heinrich Hussmann

Zweitgutachter: Prof. Dr. Detlef Zühlke

Tag der mündlichen Prüfung: 4. June 2012

Eidesstattliche Erklärung

Ich versichere an Eides Statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe. Alle Stellen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Ich versichere außerdem, dass ich keine andere als die angegebene Literatur verwendet habe. Diese Versicherung bezieht sich auch auf alle in der Arbeit enthaltenen Zeichnungen, Skizzen, bildlichen Darstellungen und dergleichen. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ingolstadt, den 28. Juni 2012 ____________________

Linshu Duan

Abstract

The human-machine interfaces (HMIs) of today's premium automotive infotainment systems are complex embedded systems which have special characteristics in comparison to GUIs of standard PC applications, in particular regarding their variability. The variability of infotainment system HMIs results from different car models, product series, markets, equipment configuration possibilities, system types and languages and necessitates enormous testing efforts. The model-based testing approach is a promising solution for reducing testing efforts and increasing test coverage. However, while model-based testing has been widely used for function tests of subsystems in practice, HMI tests have remained manual or only semi-automated and are very time-consuming and work-intensive. Also, it is very difficult to achieve systematic or high test coverage via manual tests. A large amount of research work has addressed GUI testing in recent years. In addition, variability is becoming an ever more popular topic in the domain of software product line development. However, a model-based testing approach for complex HMIs which also considers variability is still lacking. This thesis presents a model-based testing approach for infotainment system HMIs with the particular aim of resolving the variability problem. Furthermore, the thesis provides a foundation for future standards of HMI testing in practice.

The proposed approach is based on a model-based HMI testing framework which includes two essential components: a test-oriented HMI specification and a test generation component. The test-oriented HMI specification has a layered structure and is suited to specifying data which is required for testing different features of the HMI. Both the dynamic behavior and the representation of the HMI are the testing focuses of this thesis. The test generation component automatically generates tests from the test-oriented HMI specification. Furthermore, the framework can be extended in order to automatically execute the generated tests. Generated tests must first be initialized, which means that they are enhanced with concrete user input data. Afterwards, initialized tests can be automatically executed with the help of a test execution tool which must be extended into the testing framework.

In this thesis, it is proposed to specify and test different HMI-variants which have a large set of commonalities based on the software product line approach. This means the test-oriented HMI specification is extended in order to describe the commonalities and variabilities between HMI variants of an HMI product line. In particular, strategies are developed in order to generate tests for different HMI products. One special feature is that redundancies are avoided both for the test generation and the execution processes. This is especially important for the industrial practice due to limited test resources. Modeling and testing variability of automotive HMIs make up the main research contributions of this thesis.

We hope that the results presented in this thesis will offer GUI testing research a solution for model-based testing of multi-variant HMIs and provide the automotive industry with a foundation for future HMI testing standards.

Zusammenfassung

Die Mensch-Maschine-Schnittstellen (HMIs) von Infotainmentsystemen der heutigen Premiumfahrzeuge sind sehr komplexe und eingebettete Systeme. Sie haben im Vergleich mit herkömmlichen PC-Applikationen besondere Eigenschaften, insbesondere bezogen auf ihre Variabilität. Die Variabilität von Infotainmentsystem HMIs ergibt sich aus unterschiedlichen Fahrzeugsmodellen, Produktserien, Märkten, Ausstattungen, System- sowie Sprachvarianten. Die hohe Anzahl der Varianten führt zu enorm hohem Testaufwand. Modellbasiertes Testen ist ein vielversprechender Ansatz, um den Testaufwand durch die automatische Testfallgenerierung und Testausführung zu reduzieren und gleichzeitig die Testabdeckung zu erhöhen. Während modellbasiertes Testen bereits für Funktionstests häufig eingesetzt wird, bleiben HMI Tests meist noch manuell oder teil-automatisiert. Außerdem kann durch manuelles Testen eine systematische Testabdeckung nur sehr schwierig erreicht werden. Zahlreiche Forschungsarbeiten befassen sich mit dem GUI-Testen. Variabilität ist im Bereich der Software-Produktentwicklung ein immer beliebteres Forschungsthema. Ein modellbasierter Testansatz für komplexe HMIs mit Berücksichtigung der Variabilität ist allerdings immer noch nicht vorhanden. Diese Doktorarbeit präsentiert eine modellbasierte Testmethode für Infotainmentsystem HMIs mit dem besonderen Ziel das Variabilitätsproblem zu lösen. Zusätzlich bietet diese Doktorarbeit eine Basis für zukünftiges HMI-Testen in der Industrie an.

Der Ansatz in dieser Doktorarbeit basiert auf einem modellbasiertem HMI-Testframework, das zwei essentielle Komponenten beinhaltet: eine Test-orientierte Spezifikation und eine Komponente zur Testgenerierung. Die Test-orientierte Spezifikation hat eine geschichtete Struktur und ist darauf ausgerichtet, die fürs Testen relevanten Daten zu spezifizieren. Sowohl dynamisches Menüverhalten als auch die Darstellung des HMI sind die Testziele. Die Testgenerierung erzeugt automatisch Tests aus der Test-orientierten HMI Spezifikation. Das Testframework kann um eine automatische Testausführung erweitert werden. Nachdem die generierten Tests instanziert werden, ist es möglich sie automatisch innerhalb eines Testautomatisierungsframeworks durchzuführen.

Diese Doktorarbeit befasst sich mit Methoden, um die HMI-Varianten effizient zu spezifizieren und zu testen und basiert auf Ansatz für Software Produktlinien. Das bedeutet, die Test-orientierte Spezifikation ist erweitert um sowohl die Gemeinsamkeiten als auch die Spezialitäten der Varianten zu beschreiben. Insbesondere werden Strategien entwickelt, um Tests für unterschiedliche Varianten der Produktlinien automatisch zu generieren. Die Besonderheit dabei ist, dass Redundanzen sowohl für den Generierungsvorgang als auch den Ausführungsvorgang vermieden werden können. Das ist wegen den eingeschränkten Ressourcen und aus Effizientsgründen besonders wichtig für die Industrie. Die Modellierung und das Testen von variantenreichen HMIs stellen die Hauptbeiträge dieser Dissertation dar.

Die Ergebnisse dieser Doktorarbeit können hoffentlich als eine Lösung für modellbasiertes Testen der multi-varianten HMIs dienen und der Automotive-Industrie eine Basis der zukünftigen HMI Testenstandards liefern.

Danksagung (Acknowledgments)

Diese Arbeit ist während der Mitarbeit in der Abteilung Infotainmentsystem Entwicklung und Testen bei der AUDI AG als Doktorandin und unter Betreuung der Universität München entstanden. An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Promotion und Erstellung dieser Arbeit unterstützt haben.

Mein größter Dank geht an meinen Doktorvater Professor Hussmann, der mir wissenschaftliche Werte durch zahlreiche Gesprächsstunden, E-Mails und Korrekturarbeiten vermittelt hat. Sein respektvoller Umgang mit Menschen wird mir in meinem Leben immer ein Vorbild sein.

Herrn Professor Zühlke danke ich sehr für das entgegengebrachte Interesse und die Bereitschaft das zweite Gutachten zu erstellen.

Ich möchte mich besonders bei Alexander Höfer bedanken. Als Betreuer bei der AUDI AG hat er mir die Chance für eine Promotion bei Audi gegeben. Seine langjährige Unterstützung hat maßgeblich am Erfolg meiner Arbeit beigetragen. Ebenso bedanke ich mich bei meinem Abteilungsleiter während der Doktorarbeit Dieter Niederkorn, für die Schulung in der Kommunikation und Arbeitsweise im Unternehmen. Mein Dank gilt ebenso allen Kollegen und Studenten für die hervorragende Unterstützung und Zusammenarbeit.

Meine Familie hat mir während der gesamten Promotion unheimlich viel Unterstützung gegeben. Mein herzlicher Dank geht an meine lieben Schwiegereltern Petra und Dirk Brand, die mir viel Wärme und Liebe geschenkt haben. Mein lieber Dank gilt meinem Ehemann Sebastian Brand, der mir immer Zuspruch gegeben hat und mir in schwierigen und entscheidenden Phasen immer den Rücken freigehalten hat.

Mehr als nur Dank gebührt meinen Eltern. Sie haben mir in der frühesten Kindheit bereits Werte mitgegeben, die mir in meiner Ausbildung, beruflichen und privaten Leben immer zur richtigen Entscheidung verholfen haben. Sie sind zwar während meines Studiums und der Promotion nicht in meiner räumlicher Nähe gewesen, jedoch waren sie bei jeder Schwierigkeit, Freude, Spannung und Entscheidung eng bei mir. Sie geben mir unheimlich viel Energie für jede Phase des Lebens. Mein Glück und meine Erfolge verdanke ich ihnen.

Contents

Part I.

Introduction

In the development of modern vehicles, the infotainment system [54] belongs to the innovative area. In comparison to the conventional areas such as the motor, body construction and drive train, the infotainment system gains ever increasing influence on the appearance of a vehicle in the eyes of both customers and competitors [119]. Infotainment systems of current premium vehicles are highly integrated and distributed systems and provide a variety of information, entertainment and communication functions with the help of numerous electronic control units (ECUs) which are connected to each other via bus systems [86].

Figure 0.1 shows the infotainment system for the car model A8.

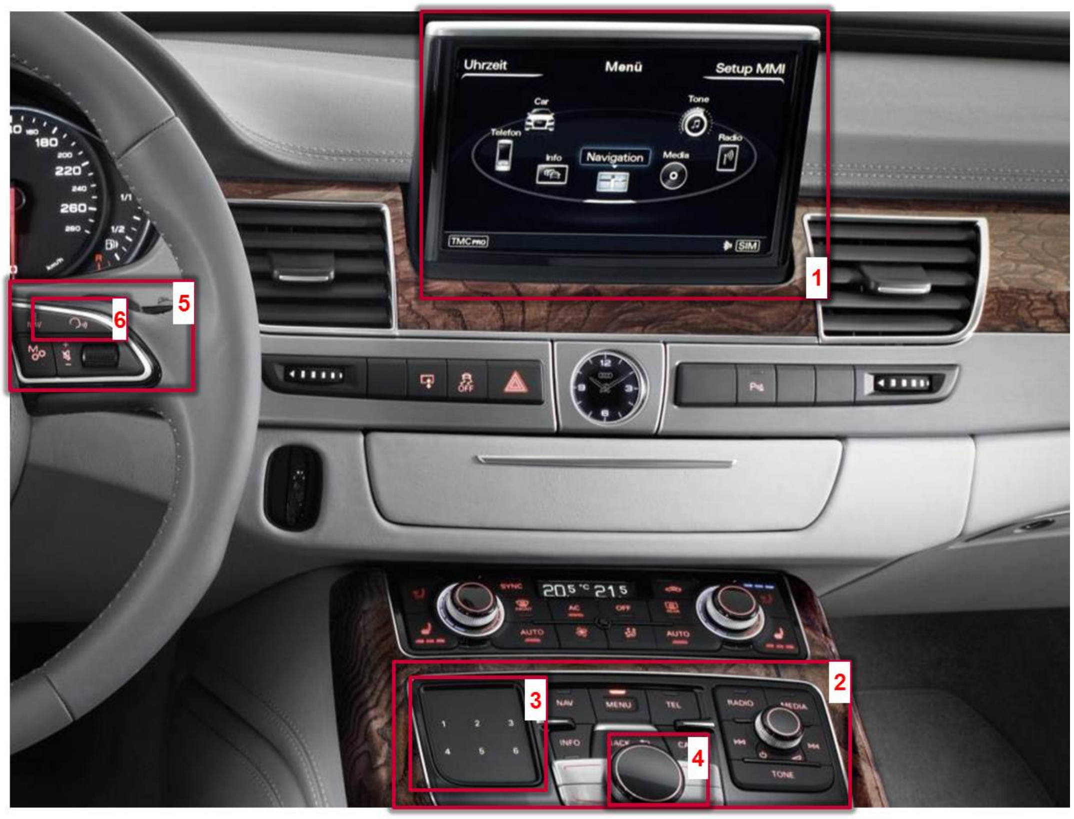


Figure 0.1.: Automotive infotainment system
1.GUI and central display 2.control unit 3.touch pad
4.turn-press button 5.multi-function-wheel 6.speech button

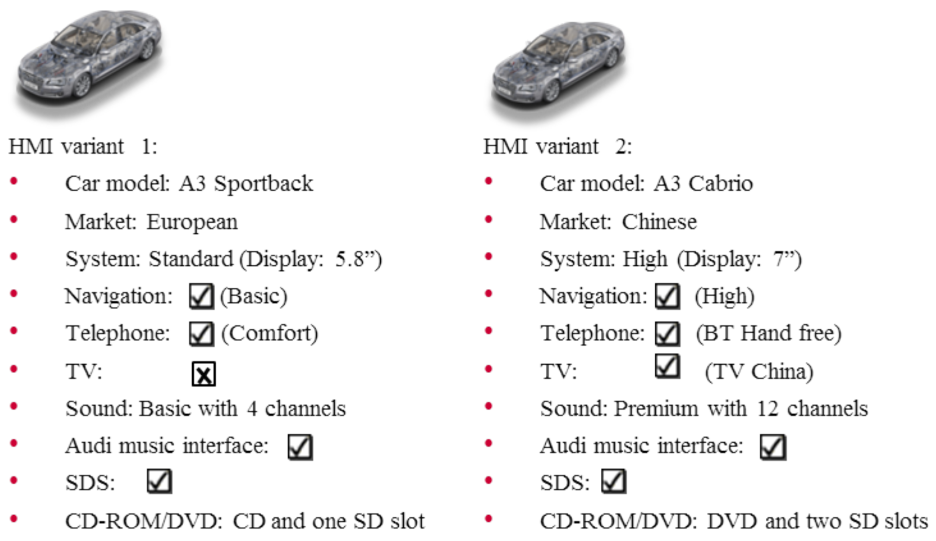
A human machine interface, abbreviated as HMI, is the composition of interfaces through which the user communicates with the machine. In the context of infotainment systems, an HMI can include (as shown in Figure 0.1) a graphical user interface (GUI) on the central display, a touch pad, buttons, a turn-press button, a multi-function wheel and speech input and output facilities. In the A8, the touch pad, buttons and turn-press button are integrated into one unit, known as the control unit. In many premium vehicles, the HMI also contains a graphical interface on the instrument cluster; this is not shown in Figure 0.1.

In spite of this general definition, the term "HMI" has a specific meaning in the context of HMI development in the area of automotive infotainment systems. In this context, the HMI does not mean the hardware interfaces as introduced above, but the software of the graphical user interface. In most of the current research in this area, the term "HMI" is a synonym for the graphical user interface of the central display [112] [45] [54].

In this thesis, the term "HMI" is assumed to have that same meaning. This means that the goal of this thesis is to test the graphical user interface on the central display. Both the graphical representation and the dynamic behavior of the HMI are focal points of this thesis. The HMI to be tested is modality-independent. This means the events to which the HMI reacts can be triggered by different input sources. Furthermore, the GUI of the instrument cluster and consequently the synchronization between the instrument cluster and the central display are not included in the scope of this thesis.

The HMI of a current premium infotainment system can be a huge system. An Audi HMI currently in development contains up to 2200 screens, 100 pop-up menus, up to 10000 texts per language and a very complex dynamic behavior. For instance, the description of the dynamic behavior can require 200 diagrams, 50 hierarchies, 2400 view states and 14000 transitions. In [45], it was declared that up to 4000 UI elements such as screens, buttons, lists and messages are used for the BMW 7-Series.

Furthermore, the HMI of a premium vehicle usually provides a large set of variants that are a result of differing car models, markets, languages and system hardware [16]. Variants can also be caused by individual combinations of functions, e.g. in some variants the navigation function is integrated; in others it is not. Figure 0.2 presents an example of two different HMI variants. In practice, more than 100 variants are possible. This variability of HMIs results in very high testing complexity.



HMI variant 1:

- Car model: A3 Sportback
- Market: European
- System: Standard (Display: 5.8”)
- Navigation: ☑ (Basic)
- Telephone: ☑ (Comfort)
- TV: ☒
- Sound: Basic with 4 channels
- Audi music interface: ☑
- SDS: ☑
- CD-ROM/DVD: CD and one SD slot

HMI variant 2:

- Car model: A3 Cabrio
- Market: Chinese
- System: High (Display: 7”)
- Navigation: ☑ (High)
- Telephone: ☑ (BT Hand free)
- TV: ☑ (TV China)
- Sound: Premium with 12 channels
- Audi music interface: ☑
- SDS: ☑
- CD-ROM/DVD: DVD and two SD slots

Figure 0.2.: An example of two HMI variants

Chapter 1.

Problem and Solution Summary

This thesis addresses the model-based testing of infotainment system HMIs with the particular aim of modeling and testing variability of the HMIs.

Current problems will be introduced from an industrial perspective as well as a research perspective. Afterwards, an overview of the solution will be presented.

1.1. Problem summary

As mentioned above, the HMI of a current infotainment system can contain up to 2200 screens, 100 pop-up menus, 10000 texts per language and a very complex dynamic behavior, the description of which requires up to 200 diagrams, 50 hierarchies, 2400 states and 14000 transitions. More than 100 variants can be caused by different car models, markets, languages, system hardware types and individual function configurations. HMI variability necessitates very high specification and especially testing efforts.

Current situation

Currently, HMI tests are performed almost entirely manually in the serial development [27] [112]. Once a test specification has been created for one project, it is usually manually adapted for later projects. In this process, defining new test cases and increasing test coverage are usually ignored due to limited time. Often, only the most important tests, which are usually a subset of the tests described in the test specification, can be executed. Manual testing is very time-consuming and expensive. First of all, it is quite difficult to achieve systematic and high test coverage via manual testing. Currently, increasing test coverage of infotainment system HMIs is a common goal of many manufacturers, especially when it comes to test coverage of foreign language systems [27]. An internal statistic of a current infotainment system project has shown that, although the HMI is only one small component out of many ECUs and bus systems in an infotainment system, it produces about 10% of the errors. This statistic is based on the internal evaluation of error tickets, which are randomly chosen from different developmental phases. This result should provide an indication of the profitability of automating HMI tests and increasing test coverage.

Problems in the industry

Model-based testing [42] [105] [62], or MBT in short, makes possible automatic test generation and systematic test coverage. In combination with automated test execution, which is usually called test automation [13], MBT provides a reduction in human resource requirements and costs. Model-based testing of application functions is well established in industry today. However, model-based testing of infotainment system HMIs is still very rare.

The most efficient method of MBT is to verify the System Under Test (SUT) against tests generated from the specification. However, most HMI specifications are still informal, even though formal specifications are more precise and can be reused for development. Efforts to establish formal specifications [37] have failed because the tools used are based on concepts designed for development and are not intended for specification. Development tools support the complexity of code generation, whereas a specification should describe the requirements in abstract form only. The complexity involved in using development tools thus leads to a failure to create formal specifications.

Tests can also be generated from the system model (Section 2.1). However, a system model is created for generating software code. It usually cannot be directly used for test generation due to a lack of test data. The necessary test data must be extended in advance if a system model is to be used for test generation. In this case, the code generator is validated against the test generator. Non-conformance between the implementation and the specification cannot be detected. Therefore, MBT is meaningful only if the test model and the system model are different.

Furthermore, an HMI testing concept that answers fundamental questions is still missing in the industry. For instance, the issues regarding which HMI errors should and can be dealt with, how to model the HMI for testing purposes, how to model variability and test different variants of the HMI, etc., have only recently been addressed [65].

Finally, variability is a new challenge for the development of infotainment system HMIs. The industry is still busy with variability development, e.g. in optimizing the development process and extending the development tools. The next step, i.e. testing under consideration of variability has not yet become a focus.

Problems in the research

The research topic of this thesis spans the domains of GUI, model-based testing and variability, in which the GUI plays a central role. Related work can be found in the 4 domains which are presented in Figure 1.1.

There are currently a number of research efforts that address the model-based testing of GUI applications [84] [79] [104] [96] [20] [89]. The GUI applications in question are mostly standard PC applications such as a calendar which contains all the necessary logic and data within the application. In comparison to standard GUI applications for PCs, an infotainment system has many special characteristics such as embedding and variability which do not exist in standard GUI applications and therefore are not considered for testing. The special characteristics of infotainment system HMI will be introduced in Section 2.2.5.

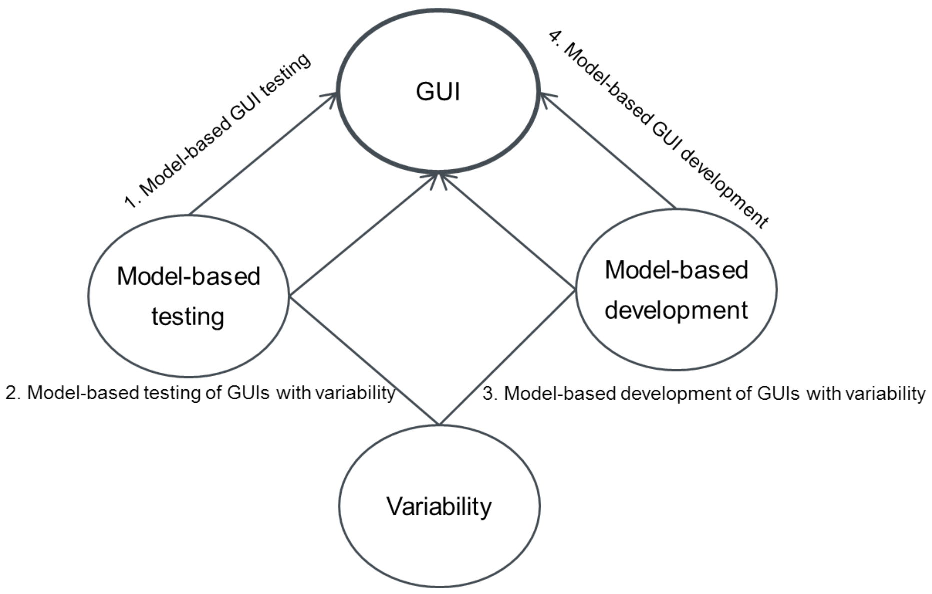

GUI
1. Model-based GUI testing
4. Model-based GUI development
Model-based testing
Model-based development
2. Model-based testing of GUIs with variability
3. Model-based development of GUIs with variability
Variability

Figure 1.1.: Related research areas

A small amount of research efforts address the model-based testing of GUIs with variability. However, many of the problems we face in HMI testing under variability do not exist in these research areas and hence have not been resolved.

There is also a small amount of research work addressing model-based development of GUIs with variability e.g. [16]. Model-based development and model-based testing face common problems, but also different problems. For instance, model-based testing involves the additional task of identifying data required for the test generation (Chapter 5) and finding a suitable test generation method and adequate selection criteria (Chapter 6) etc. In particular, if the variability of HMIs is taken into account, new specification methods must be designed and testing must address the problem of redundancies. This issue will be introduced in Part III.

1.2. Solution summary

This thesis presents a model-based testing approach for infotainment system HMIs with the particular aim of considering variability. The approach can be divided into the following parts:

1.2.1. The framework for model-based HMI testing

In this section, we propose a model-based testing framework for infotainment system HMIs. Variability is not yet considered here.

First, a statistical analysis has been done in order to identify the possible HMI errors occurring in practice. Based on this result, HMI errors which can be automatically detected in the testing framework will be clarified.

We propose two basic components for the framework: the test-oriented HMI specification and the test generation as presented in Figure 1.2.

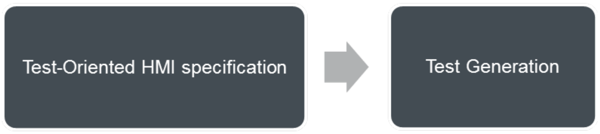


Figure 1.2.: Proposed model-based testing framework for infotainment system HMIs

The test-oriented HMI specification

A test-oriented HMI specification describes the expected behavior of the HMI with a layered structure and contains sufficient data in suitable forms to generate valid tests. The test-oriented HMI specification allows abstract specification of the HMI.

Test generation

It is not the focal point of this thesis to develop a new test generation method. A test generation algorithm is introduced in order to demonstrate how tests can be generated from the test-oriented HMI specification and what the tests look like. This test generation algorithm serves as a basis for the later generation method that takes variability into account.

1.2.2. Integrating variability into model-based testing

In this section, the variability of HMIs is taken into account. Two essential problems exist for testing under variability: how to model variability and how to generate tests for different variants without redundancies.

Modeling variability

Variability of HMIs can exist both in the dynamic behavior and the representation. Layers of the test-oriented HMI specification are extended to specify the commonalities and variabilities of different HMI variants based on the product line approach.

Test generation under variability

The test generation algorithm is extended to take account of the variabilities which are described in the test-oriented specification and to generate tests for different variants. It would be inefficient to perform a test generation for each variant to be tested. Therefore, a test generation algorithm that takes account of the variabilities is designed to avoid redundant test generation.

However, tests generated for different variants still include a large set of identical tests, which verify the commonality of different variants. A method is applied to automatically identify such tests and preserve them from redundant executions.

1.3. Organization of this thesis

This thesis comprises four parts as presented in Figure 1.3.

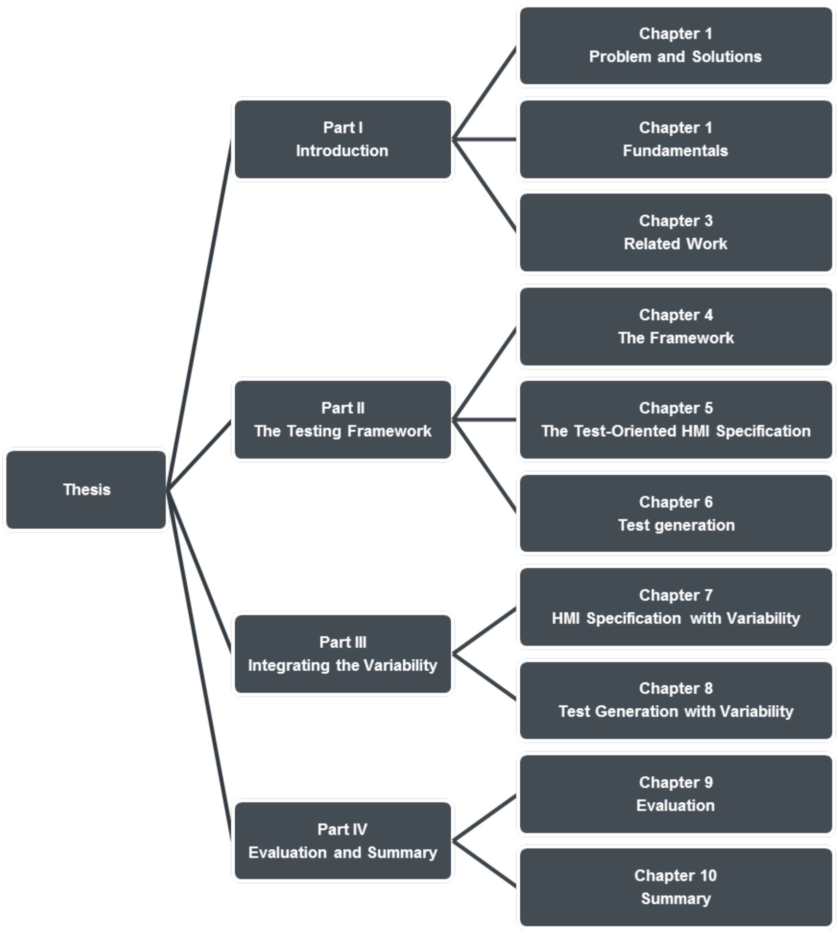


Figure 1.3.: Organization of this thesis

The first part includes Chapter 1 to 3. Chapter 1 has introduced the problems and solutions up to the present time. Chapter 2 gives an introduction to the fundamentals in the domains of model-based testing and automotive HMIs. Chapter 4 presents some of the related work, compares this with the proposed approach and explains why existing approaches are not suitable for testing infotainment system HMIs. References to other relevant work will be distributed throughout the later chapters due to the diversity of the sub-topics.

In the second part, we demonstrate the proposed testing framework. Here variability is not yet considered. In Chapter 4, components of the framework and the testing goals are presented. This thesis focuses on two types of tests: menu behavior tests verifying the dynamic behavior and screen tests verifying different classes of screen contents. Chapter 5 describes the requirements of the test-oriented HMI specification. Chapter 6 demonstrates a test generation algorithm and generated tests.

Part III focuses on variability. Chapter 7 introduces how to model variability. For modeling variability, each layer of the test-oriented HMI specification introduced in Chapter 7 will be extended. Chapter 8 demonstrates a test generation method that considers variability and generates tests for different variants without redundancies. Although redundant test generations could be avoided, generated tests for different variants still contain identical tests, which verify the commonality of these variants. At the end of chapter 8, a method of avoiding redundant executions of such tests will be introduced.

Part IV includes the evaluation and summary. Chapter 9 evaluates the modeling and testing concept for variability from the efficiency perspective. First, a mathematical plausibility analysis is demonstrated in order to show that the proposed modeling and testing concept is profitable based on a practical HMI project. Second, a general discussion is led. Factors are identified on which the efficiency improvement of the proposed approaches is dependent. Also, worst cases are identified in which it is not profitable to use these approaches. Finally, an demonstration of efficiency improvement is given using concrete data. Chapter 10 gives a summary and an overview of future work using the perspectives of the research and the industry.

Chapter 2.

Fundamentals

This thesis proposes a model-based testing approach for infotainment system HMIs. This chapter introduces the fundamentals of the domains of MBT and automotive HMI.

2.1. Model-based testing

Many different definitions of model-based testing (MBT) can be found in literature.

In [75], Whittaker and El-Far have given the following definition :

Model-based testing is [...] an approach that bases common testing tasks such as test case generation and test result evaluation on a model of the application under test.

In [87], MBT is defined as:

MBT approaches help automatically generate test cases using models extracted from software artifacts.

The developer of the UML2 Testing Profile (UTP) has given the following statement about the MBT in [9]:

Model-based testing requires the systematic and possibly automatic derivation of tests from models. In our case, UML is the language for specifying models and UTP the formalism to describe the derived tests.

The most important aspect of MBT is to automatically generate tests from a model, which formally describes the expected behavior of the *System Under Test* (SUT).

2.1.1. Variants of MBT

In [105], six different variants of MBT are introduced. We display the two most representative of these in Figure 2.1.

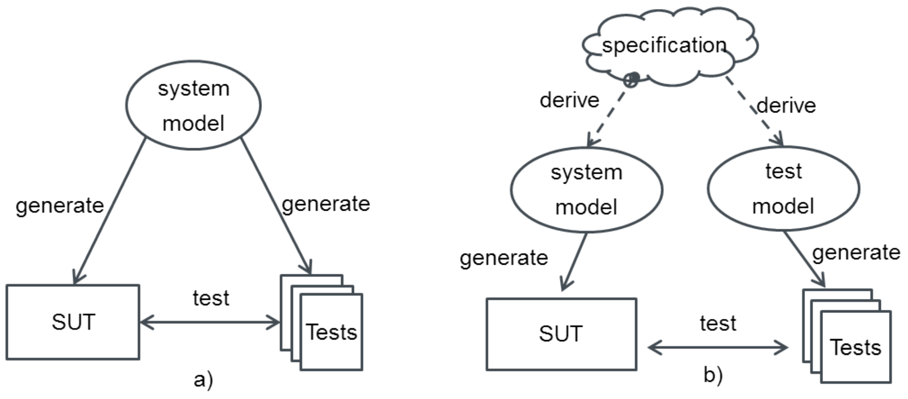


Figure 2.1.: Two possible variants of MBT (Source: [105])

A system model describes the behavior of the SUT, whereas a test model describes the features of the SUT which are to be tested. Software code is generated from a system model and tests are derived from a test model. In the variant a), the test model is also the system model. The advantage is that only one model need be created. However, testing is of very limited use here, since both the SUT and the tests are fully generated from the same source (the system model).The SUT cannot be verified against the specification. Therefore, non-conformance between the expectation and the implementation cannot be detected. In this variant, only the code generator is verified with the test generator. This problem does not exist in the variant b). In this variant, the specification is informal. Developers create a system model according to their understanding of the specification and generate system software from it. Testers create a test model, which formally describes the expectation of features to be tested. Tests are automatically generated from this test model and executed on the SUT. In this way, the SUT can be compared to the expectation described in the test model. Certainly, the specification can contain errors and testers can also make mistakes and produce errors in the test model. However, this is a general problem with MBT approaches. This variant is an overhead for the industry, since two parallel models must be developed.

In this thesis, we don't define which variant should be applied for testing the HMI. It is only necessary to define which data must be specified in the test model in order to ensure the generation of valid tests and the performance of the intended tests. The proposed approach can be realized with different MBT variants. Figure 2.2 presents one possible variant:

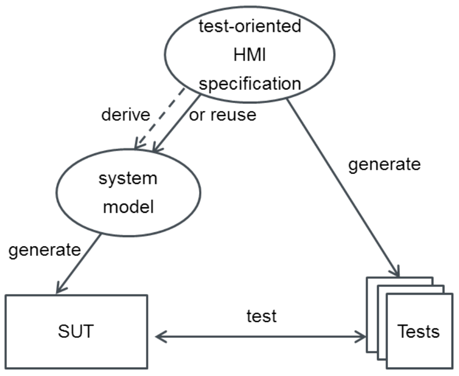


Figure 2.2.: A possible variant for the proposed MBT approach

The test-oriented HMI specification presented in Figure 2.2 is a formal HMI specification which describes the expectation of the SUT and contains sufficient data required for the test generation. In this variant, the system model can either be derived from the test-oriented HMI specification just as from an informal specification, or it can be based on it but developed further. This means that the formal specification is used as the basis for the development of the system model. Developers must extend data which is needed for the code generation into the test model, e.g. conditions including variables which can be set by underlying applications during runtime. In addition, abstract descriptions must be extended into concrete implementations, e.g. the UI elements. This usually requires a tool which allows the creation of both abstract HMI specifications and HMI development models and a well-defined work process, since both the test specifiers and developers work on the same model. The advantages are that part of the specification contents can be reused for development but the test data is separated from the development data nevertheless. In this way, the HMI implementation can be verified against the expectation.

2.1.2. Test generation

Test generation is the process in which tests are generated from the test model.

Different test models can be used to describe different features of the SUT. For example, class diagrams [15] [3] and object diagrams [15] [3] can be applied to describe static structures. Petri-nets [21] and timing diagrams [15] [3] allow one to describe parallel and real-time behavior. Use case diagrams [15] [3], environment models [125] and functional models [36] can also be used as test models. Since the goal of this thesis is to describe the dynamic behavior of the automotive HMI, we shall only discuss models describing dynamic behavior at this point: activity diagrams, state diagrams and interaction diagrams [69] [3]. UML statechart is a very popular format for state diagrams in the automotive HMI area. It will be introduced in Section 2.1.5.

The most important aspect of test generation is the coverage criteria [4] [90] [76]. Coverage criteria are used as selection criteria by the generation algorithm during the test generation. They are also used to measure the quality of the test generation. Which coverage criteria are used depends on the specification language of the test model and the testing purposes. [105]

has provided an overview of conventional coverage criteria in conjunction with an activity diagram and state diagram which are very widely used specification languages for describing dynamic behavior in the infotainment system domain. Figure 2.3 presents a subset of the coverage criteria given in this overview: the structural coverage criteria [42]. Besides structural coverage criteria, there are also functional and stochastic criteria [42] and generation methods based on statistics, requirements and explicit rules [105].

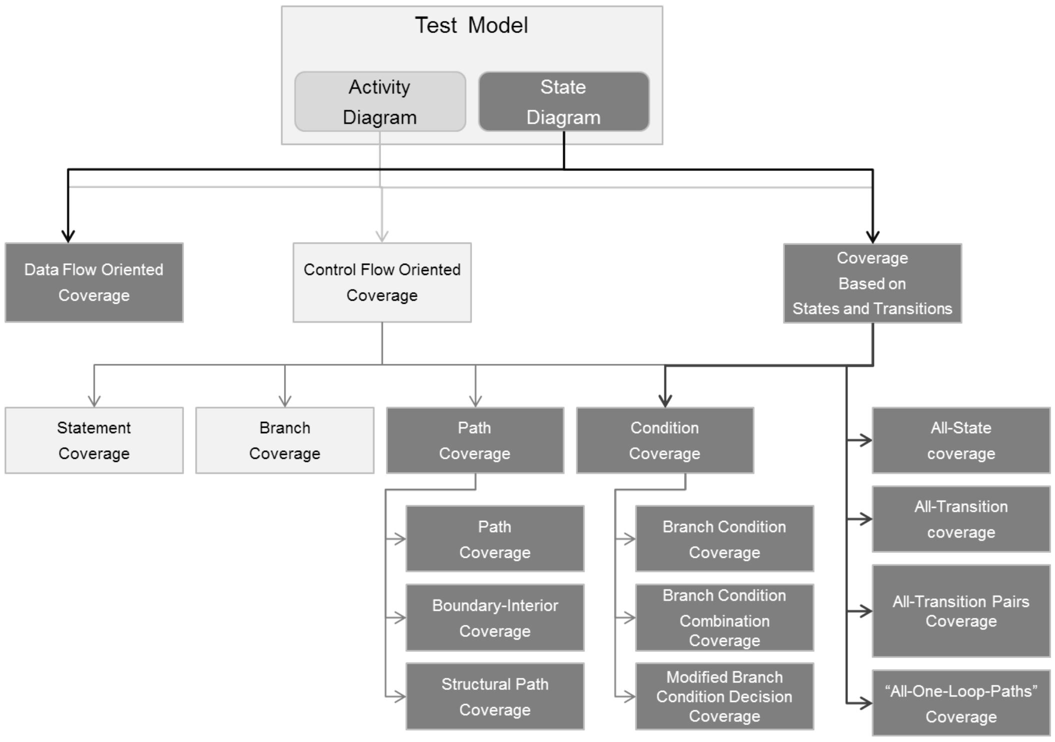

Test Model
Activity Diagram
State Diagram
Data Flow Oriented Coverage
Control Flow Oriented Coverage
Coverage Based on States and Transitions
Statement Coverage
Branch Coverage
Path Coverage
Condition Coverage
All-State coverage
Path Coverage
Boundary-Interior Coverage
Structural Path Coverage
Branch Condition Coverage
Branch Condition Combination Coverage
Modified Branch Condition Decision Coverage
All-Transition coverage
All-Transition Pairs Coverage
"All-One-Loop-Paths" Coverage

Figure 2.3.: Conventional coverage criteria for state and activity diagrams (Source: [105])

Of these structural coverage criteria, the **transition coverage**, **path coverage** and **state coverage** are relevant for the test generation method in this thesis. The transition coverage and state coverage are only applicable for state-based test models.

"*Transition coverage requires choosing test cases in such a way that all transitions of the specification are covered.*" ([42]).

"*The strongest coverage criterion is the path coverage criterion. This criterion is satisfied by a test suite if and only if for any possible path in the model, the test suite contains at least one test case which enforces an execution of this path in the implementation. Path coverage is in general impossible to achieve and impractical for real life testing.*" ([42]).

In [105], it was defined that each state in the model must be covered in generated tests for **state coverage**.

The transition coverage is stronger than the state coverage. If all transitions are covered, then each reachable state is also covered. However, if a state can have more than one incoming transitions, it is possible that the state is covered via one of these transitions but the other transition is not visited.

2.1.3. Automated test executions

MBT can also be combined with automated test executions; this is also called test automation [35] [61]. Test automation is the process in which tests are automatically executed in a test automation environment. In connection with MBT, automatically generated tests should be automatically executed.

Test models are usually abstract. This means they only contain events triggering transitions and do not contain concrete user data such as a specific phone number to dial during test execution. Therefore, generated tests are still not automatically executable. The process of mapping abstract tests to executable tests is called test instantiation [13].

2.1.4. Current situation of MBT in practice

Today, model-based testing is well established in the automotive area for function tests [19]. At Audi, infotainment system functions too are currently tested using model-based and automated tests. However, HMI tests are performed almost entirely manually.

As explained in previous work [28], a function test verifies e.g. whether a phone contact can be correctly added into the address book. To do this, a tester can create a test model which describes the behavior, i.e. that a contact can be added into the address book, and defines the expectation, i.e. that the added entry is subsequently available in the address book. A generated test contains the steps required to add a contact entry, followed by the verification of the added entry. The available entries in the address book can be obtained e.g. from the database or the bus system. In the automotive domain, EXAM [64] and MODENA [URLh] are widely used testing frameworks for car functions and infotainment system functions.

In comparison to function errors, HMI errors are very varied: an error can occur in menu changes, in the graphical representation, in widget behaviors etc. In Section 4.2, more than 10 types of HMI errors occurring in practice will be introduced. Current HMI industry testing frameworks are used for component tests by the suppliers who develop the HMIs. This means the HMI is tested as a single component without any connections to the other components of the infotainment system, e.g. underlying applications and bus systems. Most of these HMI testing frameworks are based on the capture/reply method: while a tester manually executes the tests, the action sequences are captured which will later be executed on the HMI. Other available HMI testing frameworks allow the manual creation of test scripts which are also composed entirely of user actions. This means that currently available HMI testing frameworks focus only on the testing of user action sequences. Component tests are usually very limited. The problem is that it is not possible for HMI suppliers to perform integration tests because the different components of the infotainment systems are developed by different suppliers and the final integration is usually done by the manufacturer. Furthermore, manual capture or creation of tests cannot achieve systematic test coverage. An HMI testing framework is still lacking which allows users to create an HMI test model describing the varied features of the HMI, supports automatic test generation and hence enables specific forms of test coverage to be achieved systematically.

2.1.5. UML statechart

UML statechart (UML SC) [106] [26] is a very widely used specification language for describing dynamic system behaviors. A SC model describes the possible states of a system and possible flows between these states. State changes are triggered by events via transitions which can be labeled with guard conditions and actions. If the event of a transition has been triggered and the guard conditions are fulfilled, then the actions will be performed and the next state is achieved. SC diagrams allow hierarchies.

UML SC allows the creation of semi-formal or formal specifications by explicitly and exactly defining the used notation, syntax and semantics of the UML SC. For instance, the system model, from which software code is to be automatically generated, must be a formal model. In Chapter 5, the formal definition of UML SC will be introduced.

2.2. Automotive HMIs

As already explained in the introduction, in the context of model-based testing for automotive infotainment system HMIs, the term "HMI" is synonymous with the "graphical user interface" presented on the central display.

Memon, in his thesis [84], defines the GUI as follows:

> *A GUI is the front-end to underlying code, and a software user interacts with the software using the GUI. The user performs events such as mouse movements, object manipulation, menu selections, and opening and closing of windows. The GUI, in turn, interacts with the underlying code through messages and/or method calls.*

An infotainment system HMI is a particularly complex GUI. It is the front-end of the infotainment system composed of different ECUs and bus systems. It reacts to user events triggered via different input facilities and interacts with the ECUs of the infotainment system via underlying applications. An infotainment system HMI contains both the graphical representation, i.e. screens and their contained UI elements, and the dynamic behavior, i.e. the dynamic changes between different screens.

2.2.1. The graphical representation of an HMI

Figure 2.4 and 2.5 show two screens: ‘Main’ and ‘NavAdrInput’. Screen ‘Main’ allows users to access any one of the functions provided by the infotainment system, such as radio, telephone and navigation. When the navigation function is selected, the screen ‘NavAdrInput’ allows users to input a destination and start the navigation guidance.

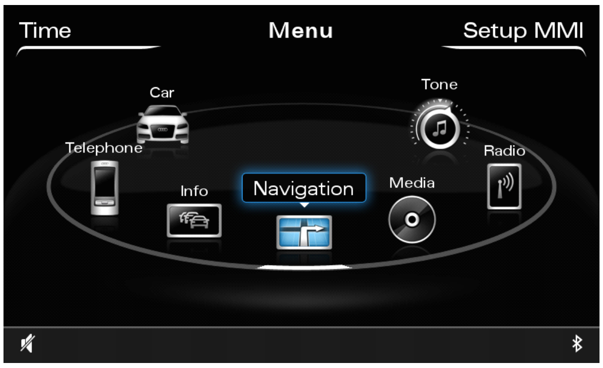
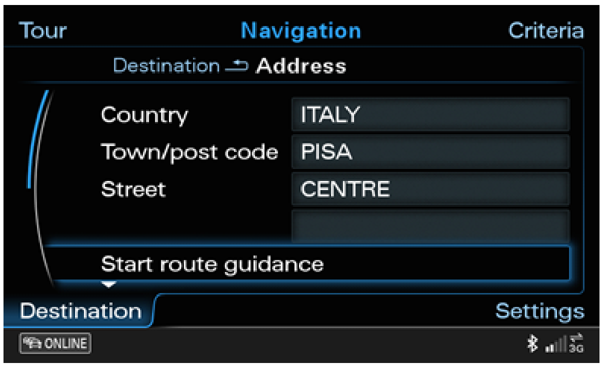


Figure 2.4.: Screen ‘Main’



Country ITALY
Town/post code PISA
Street CENTRE

Figure 2.5.: Screen ‘NavAdrInput’

Figure 2.6 presents a subset of the widgets contained in the screen ‘NavAdrInput’.

UI elements contained in a screen are usually called widgets. A screen is also a widget. Widgets such as the title and soft keys present static text to users. These widgets usually have a text label ID as a property. The contents of such text labels are usually externally defined, since current HMIs usually have more than one language and translations or changes of these texts must be performed outside of the virtual HMI. The widget status bar contains sub-widgets which are icons. The widget scroll list is a complex widget, and contains further complex widgets such as the scroll bar and several rows of options. This list widget has complex behavior. The user can scroll between the rows; during this operation the focused row must be highlighted. If there is at least one page after the current page, the list must display an arrow at the bottom pointing downward. If there is a previous page, the list must present an arrow at the head pointing upward. The scroll bar must also be able to calculate the position

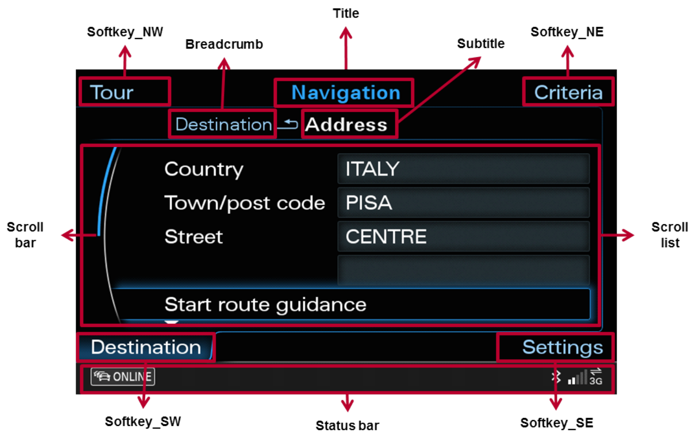


Figure 2.6.: Some widgets contained in the screen ‘NavAdrInput’

of the current page in order to correctly indicate the position. The first row of the list contains a text widget presenting static text and a text widget presenting dynamic text. Dynamic text is not defined for the HMI, but is provided by the user during runtime.

In practice, different kinds of errors can occur in the screens of the HMI. These errors have been identified in this thesis and will be introduced later.

2.2.2. The dynamics of an HMI

An infotainment system HMI usually has a very complex menu behavior. A menu change can be triggered by a user action or a message sent by underlying applications. Figure 2.7 presents the menu behavior of entering a navigation destination and starting the route guidance.

The first screen is the screen ‘NavAdrInput’ which was presented in Figure 2.5. It allows users to enter a destination. As shown, the user can first choose to input a country (step 1) and then select a country from the country list. The user is finally led back to ‘NavAdrInput’ (step 2). From here the user can choose to input a city (step 3), which must be entered with a speller. After a city has been defined, the user is led back to ‘NavAdrInput’ (step 4). From here, similarly to entering a city, the user can enter a street (step 5 and 6). Now the route guidance can be started (step 7). As soon as the calculation is done, the map screen is presented to the user (step 8). This is only one of several possible ways of entering a destination. For example, it is also possible to enter a street directly after a country.

Automotive HMIs usually also contain a number of pop-up windows, which are used to present temporary or spontaneous information to users. For instance, the pop-up indicating an empty tank is initialized by the underlying application; the pop-up presenting the sound volume scale appears when the user changes the sound volume. Pop-up screens are only displayed in front of the actual screen. They don’t change the state of the main menu behavior. Since pop-ups are not in the focus of this research, they will not be discussed further at this point.

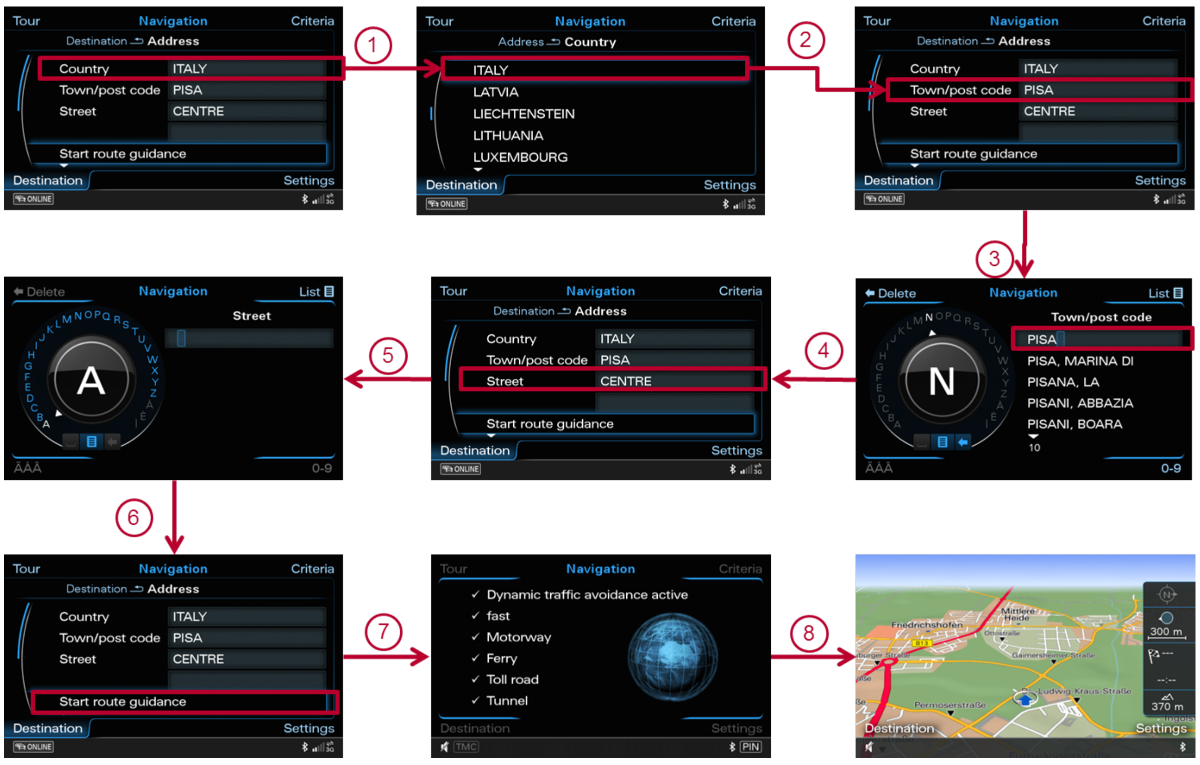


Figure 2.7.: An example of the HMI menu behavior

2.2.3. Development process of an HMI

Stand-alone PC software contains the complete logic and data within the application and has a simple and continuous development process. In comparison, an infotainment system HMI communicates with a variety of ECUs via underlying applications, which are developed by different suppliers. This leads to a much more complex development process than for a standard stand-alone PC application.

The evaluation of the current HMI development processes at many German manufacturers and suppliers has shown that the specification, development and testing steps in an HMI development process are usually discontinuous [46]. Commonly, the manufacturer creates an informal HMI specification using generic tools and formats such as PDF and Visio. The HMI supplier develops the HMI according to this specification. Most HMI suppliers today develop the HMI with model-based development tools such as EB GUIDE Studio [URLf] and VAPS XT [URLi]. The result of the development is called a **development model**, which contains both the concrete screens and usually state machines describing the dynamic behavior. Together with its HMI framework, the HMI software can be automatically generated via a code generator from the development model. Different suppliers work together for the integration of the different components of the infotainment system. For instance, the HMI supplier works with the supplier providing the hardware for the HMI, while the hardware supplier must work with suppliers producing different ECUs. Finally, the infotainment system software including the HMI software is delivered to Audi and is ready to be tested. This shows that the specification, development and testing are disconnected in the development process and are performed by different teams, usually using different tools.

2.2.4. Variants of HMI

HMIs of premium vehicles are usually provided in different variants. Variants can be necessitated by differing car models, markets, languages, system hardware types and also individual combinations of functions. Figure 0.2 presents an example of two different HMI variants. In practice, more than 100 variants are possible. Different variants share a large set of commonalities. For instance, the same screen of different variants contains a set of common widgets. It contains only one or a few additional widgets for some of the variants. The menu behaviors, too, of different variants are very similar. Variability causes enormous testing complexity.

2.2.5. Conclusion: Automotive HMI vs. standard PC applications

Testing the automotive HMI faces more challenges than testing standard PC applications due to its special characteristics.

Firstly, the HMI is an embedded system, which communicates with the ECUs via underlying applications. The dynamic menu behavior and the represented contents are dependent on these underlying applications. Secondly, an infotainment system HMI is based on the concept of screens and can include more than 2000 screens. Screens and screen changes must be especially considered for testing. Thirdly, the HMI is usually provided in several languages. Therefore, concrete texts cannot be directly included in the HMI. Furthermore, the testing approach must be realizable for the complex and noncontinuous development process. Finally, an infotainment system HMI has a large set of variants. Variability must be taken into account when modeling and testing the HMI.

Chapter 3.

Related Work

The research presented in this thesis focuses on the model-based testing of automotive HMIs, which have a special characteristic: variability. As presented in Figure 3.1, the topic spans the domains of GUIs, model-based testing and variability, in which the GUI plays the central role. In relation to the combination of GUI and model-based testing (model-based GUI testing, which is area 1 in Figure 3.1), a large set of research results and techniques can be found. GUIs which have variability are also research topics. Here, we will focus on the model-based approaches (area 2 and 3 in Figure 3.1), since model-based development is already in heavy use for car infotainment systems. There are various research efforts in the domain of model-based development of GUIs (area 4 in Figure 3.1). The commonality between the model-based testing and model-based development of GUIs is that both domains rely on the same basis: the GUI modeling. Related work in model-based GUI development will be introduced in Section 5.4 in the context of GUI modeling.

This section introduces related work in the areas 1-3 as presented in Figure 3.1, identifies the commonalities and differences to our approaches and clarifies why the existing research results cannot be used to resolve our problems. Due to the diversity of the subtopics covered, part of the related work will be presented in later chapters in the context of the respective subtopics.

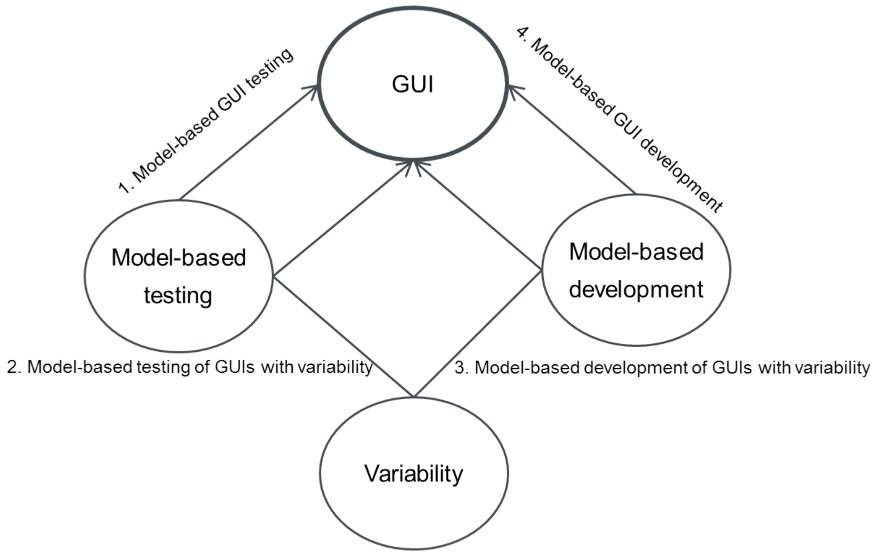

GUI
1. Model-based GUI testing
4. Model-based GUI development
Model-based testing
Model-based development
2. Model-based testing of GUIs with variability
3. Model-based development of GUIs with variability
Variability

Figure 3.1.: Related research areas

3.1. Model-based GUI testing

A number of research efforts are currently focused on model-based GUI testing (area 1 in Figure 3.1).

A number of approaches focus on the testing of Finite State Machines (**FSM**). Here, **event-based modeling** and **state-based modeling** can be distinguished.

In some early approaches of Atif Memon, [80] [81] and [72], the automatic test case generation for GUIs is based on the **planning** technique: for a "*given a set of operators, an initial state and a goal state*", a sequence of operators can be generated which lead the system from the initial state to the goal state. The precondition is that the GUI is described as a flow of allowed operators. Atif Memon's later approaches are mainly based on **event-flow graphs** which are also FSMs. He has developed a comprehensive GUI testing framework in the doctoral thesis [84]. The GUI testing framework contains a GUI model as the central component which is composed of event-flow graphs. In order to obtain the GUI model, the hierarchical structure of the GUI is exploited and the most important event sequences are identified. The framework also includes a test coverage evaluator, a test case generator, test oracles, a test executor and regression testers. Hierarchical and event-based coverage criteria are used for the test generation. By defining tests as event sequences, the thesis aims to test varied combinations of user action sequences. In [83], several adequate event-based test coverages were introduced for test generation from event-flow graphs. In this later work [79], the event-flow graph is refined. Atif Memon later separately addressed the regression tests of GUIs in [82], the GUIs of world-wide-webs in [123] and test oracles of GUIs in [124]. In [12] Fevzi Belli proposed to model the GUI in terms of the allowed action sequences with a finite state machine. Each state represents an action which can be performed with the GUI. In order to generate tests, the FSM should be converted into an equivalent regular expression: "RegEx". It was explained that although the test generation from a FSM can be performed efficiently, RegExs offers some essential advantages, e.g. that well-known algorithms such as event algebra can be used to reduce the complexity of the RegEx.

Event-based test models are not intuitive for specifying infotainment system HMIs in comparison to state-based models. Infotainment system HMIs have a special characteristic: they contain a large number of screens between which the user can switch. Each screen is intuitively regarded as a state of the HMI, since switching from one screen to another screen is similar to leaving one state and entering another state. Event-based test models would have an acceptance problem in the automotive HMI domain in which the UML SC is already widely established in HMI specification and development.

The approach in [108] is also based on FSM. In this approach, one or a set of states represent the complete user interface in a way similar to the approach in [84]. This leads to the drawback that any further improvement or change of the system is inefficient, since many states must be adapted in the model. Furthermore, FSMs have the problem of the state explosion. A comparison of state-based modeling and event-based modeling can be found in [126].

In [121] and [122], the authors address the challenge that the GUI can contain a large number of states and complex GUI dependencies. The authors have presented a different FSM model that divides the complete state space into different sub-state machines based on *user tasks*. User tasks are action sequences which the user can perform with the GUI. They can be identified by the tester as a *responsibility.* For each responsibility, the tester identifies a machine model: the *Complete Interaction Sequences* (**CIS**) which is then used for the test generation.

In [63] and [62] Kervinen has proposed to separate the specification of the business logic and the GUI details. An *action model* is used to describe the business logic which is based on the Labeled Transition System (**LTS**). With the help of a *refinement machine* which is also based on the LTS, each action is mapped to an event which can be performed with the GUI. In order to generate tests, a composite LTS must be created by replacing the actions in the action model with the help of the corresponding refinement machines. This LTS is used to generate test cases via a test generation tool named TEMA.

Many approaches are based on **Petri nets**. The authors of [104] aim to test the structural representation of GUIs which is specified by Hierarchical Predicate Transitions Nets (**HPrTNs**). In this work, the original coverage criteria proposed for HPrTNs have been extended. The approach in [97] also proposes to model the GUI based on Petri nets. The focal point of this work is GUI design and automatic code generation.

The approaches introduced above propose the use of different specification languages: FSM, LTS, Petri nets and their variants. The common drawback of these approaches is that the proposed languages have a scalability problem when specifying huge and complex systems such as infotainment system HMIs. In particular, the approaches in which each state of a widget also makes up a state of the GUI such as [108] cannot be applied to infotainment system HMIs which have more than 2000 screens; a large set of these screens have intelligent widgets which have their own behavior.

In [89] it was proposed to model the GUI based on Hierarchical Finite State Machines (**HFSMs**). A test generation method has been introduced which directly generates tests from the HFSM.

Model-based testing of GUIs can also be based on **UML diagrams**. For instance, in [95] a subset of UML diagrams is extended in order to model the GUI. Use case diagrams and activity diagrams are used to describe the purpose and usages, class diagrams are used to describe the structure of GUI windows, and state machine diagrams are used to describe the behavior. In the testing process proposed in this work, the visual model is automatically transformed into a formal specification language, Spec#, according to some rules. The obtained formal model is then refined and completed with method bodies that have been previously defined in Spec#. In this way, an executable model is obtained. Test cases are then automatically generated from this model via the *Spec Explorer* which was developed by Microsoft. With the help of an existing GUI mapping tool, abstract user actions can be associated with concrete actions. Finally, test cases are automatically executed on the SUT.

HFSM and UML used in the approaches above have improved scalability. However, these approaches, and in fact all approaches introduced so far, only address the testing of user action sequences, which is entirely adequate for testing simple standard PC applications. Infotainment system HMIs are not only characterized by complex menu behavior as explained above but also complex representation as well as multimedia applications [100]. The representation of infotainment system HMIs and multimedia applications includes textual contents (usually in several languages), visual contents, intelligent widgets and often also interrelations between different states of these listed components. In our research, we have identified 12 types of

HMI errors which have different sources and usually different symptoms (Section 4.2). Therefore, both the menu behavior and the representation should be separately and extensively tested. Modeling and testing the allowed user actions alone cannot meet the needs of testing infotainment system HMIs.

The **NModel** [55] which is developed by Microsoft Research is used in several approaches to describe the GUI. It is a model-based testing framework and analysis tool for C# programs. It allows the user to formally describe the expectation of the SUT based on an FSM. The conformance between the expectation and the SUT can be automatically determined by the *conformance tester*. In [20] the original idea is applied to test AHLTA-Mobiles. Due to the drawback that the original NModel requires actions in the model to be bound to methods in C#, [96] has extended NModel "*by adding the capability to gather information about the physical GUI objects that are the target of the user actions described in the model, and automatically generate a .Net assembly with methods that simulate those actions upon the GUI application under test*". In common with many approaches introduced above, the NModel approach has the scalability problem and only allows the specification of dynamic behavior. A further drawback of that approach is that the NModel framework was originally developed for C# applications. The test initialization and executions inside the framework are unsuitable for infotainment system HMI tests.

In [112] the authors introduced a model-based testing approach specifically for infotainment system HMIs. A rapid HMI prototyping framework **FLUID** is proposed in which a formal HMI specification can be created. The idea is to automatically derive a prototype from the formal specification and then run the HMI implementation and the prototype synchronously in order to compare their behaviors. The architecture of FLUID has a layered structure. According to the top-down order, it contains the *user interface layer*, *dialog layer*, *application layer*, *service layer* and the *bus abstract layer* as presented in Figure 3.2.

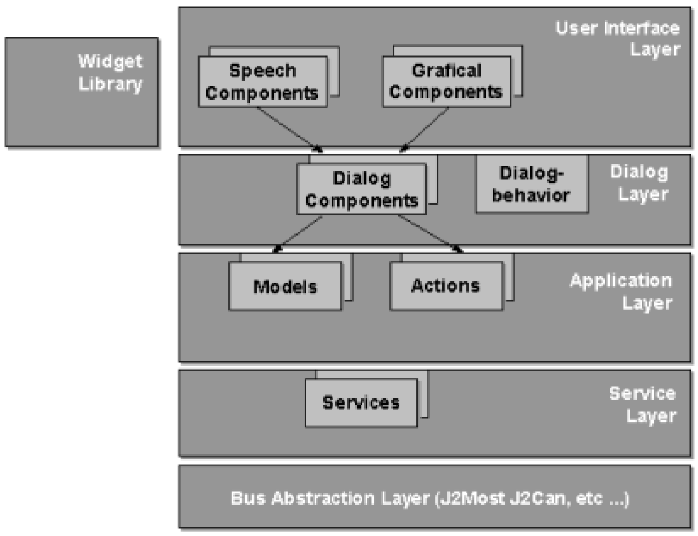


Figure 3.2.: FLUID architecture (original figure from [112])

In the user interface layer, concrete screens are created with widget libraries which must be programmed in advance. This layer is divided into available modalities, e.g. graphic, speech and haptics. The bus abstraction layer provides access to the bus systems. The service layer encapsulates application logic which is defined in the application layer and the bus messages and values which are defined in the bus abstraction layer. Application logic is described with models and actions. The models contain data obtained from bus systems. The actions which can be performed with the GUI use the contained data of one or some of these models as parameters. The dialog model maps the models and actions to user interfaces. The long-term goal is to generate tests and create prototypes from the same specification and compare the behavior of the SUT with the behavior of the prototypes. The specialty of this MBT testing approach is that the formal specification is not an abstract specification as in classical test models (Section 2.1.1). It contains concrete user interfaces, application logic and bus interfaces, and is ready for the automatic derivation of executable prototypes. Actually, this formal specification contains similar contents to a development model (Section 2.2.3). This means that creating such a specification in addition to the development model requires enormous modeling efforts and is not realistic for serial development. In fact, the same idea has already been attempted at Audi. It failed firstly because of the modeling efforts and secondly because of the high complexity and huge manual efforts required to inject the runtime messages and values into the prototype in real time. First of all, a test model from which tests can be generated definitely must contain different data to a development model from which a prototype or a SUT is generated. Testing data is required in order to ensure valid tests. This is not addressed by the FLUID framework. In particular, variability is not addressed in that work.

A GUI testing framework has been proposed in [2] which requires "*the least user involvement*". In that framework, the GUI test model describes the GUI elements in a tree structure based on the XML format and is parsed from the SUT **assembly** at runtime. Afterwards, tests can be generated from the GUI test model based on different generation algorithms. In order to verify the SUT behavior, the SUT can be simulated and the outputs are compared with the generated test cases. One drawback of this approach is that the code generator is verified against the test generator, as explained in Section 2.1.1. Furthermore, the test model is automatically generated by extracting properties and controls from the application assembly at runtime in order to reduce user involvement. However, it may require the GUI implementation to observe some rules (such as that the properties and methods to be tested must be public) and certain interfaces must be available for extracting information at runtime. It limits the applicability of that framework for infotainment system HMIs, since different generations and types of infotainment systems can have different HMI implementation bases and the required interfaces cannot be always ensured.

Tests can also be generated from models describing the user behavior. Such models are called **Novice User Tests** [60]. In this approach, a test expert first generates a sequence of user actions for a given task. Generic algorithms are then applied in order to modify and lengthen these sequences. This process is called mimicking a novice user. The drawback of this approach is that it requires intensive manual work. Furthermore, systematic test coverage of the SUT cannot be achieved, since the model is not created to describe the SUT behavior but only user behavior which may be merely a subset of the allowed SUT behavior, and because the sequences which are to be modified are also created by testers. MBT offers systematic test coverage via the automatic test generation based on certain coverage criteria. This is not applied in this approach.

The approaches introduced so far propose different model-based specification languages and model the GUI from different aspects. Coverage criteria are applied which are adequate for the respective language and aspect. The commonality between the approach of this thesis and those introduced so far is that all of them engage in modeling the GUI and generating tests from the model which is the basic principle of MBT. However, all the approaches introduced up to now address standalone PC applications which contain the complete application logic and required data within themselves. Applications which can communicate with background systems, databases or underlying applications have not been the focal point of current GUI testing research. For instance, the behavior of an infotainment system HMI is dependent on runtime data and the HMI must interact with underlying applications in order to provide complete functions to the user. The challenge of testing such embedded GUI applications consists of specifying the HMI behavior, which is dependent on the runtime data and the logic underlying applications, even though both data and logic are unknown or not available during the specification.

Moreover, an infotainment system HMI is usually provided in numerous variants which share a large set of commonalties. Testing variants implies new challenges which are not addressed in the GUI testing approaches listed above. For instance, modeling and testing each variant separately would require enormous modeling and testing efforts. Therefore, a solution for testing under variability especially for complex and embedded GUIs is required.

There are also GUI testing approaches which are not based on models e.g. the **capture/reply** concept ([49] [78]) and the **symbolic execution** ([40]). In [19], **computer vision** is proposed for testing GUIs. In this approach, "*testers can write a visual test script that uses images to specify which GUI components to interact with and what visual feedback to be observed*". Test scripts can also be generated during a manual demonstration by capturing both the input events and the displayed screen images. Capture/reply and computer vision are very useful methods and already in use in the domain of infotainment systems. Our research aims at achieving a systematical test coverage and automatic test generation which are made possible by MBT.

3.2. Model-based testing of GUIs with variability

In recent years, product line approaches have become ever more focused both in the area of model-based development [1] [115] [116] and of model-based testing [93] [94] [120].

Model-based testing of GUIs with variability (area 2 in Figure 3.1) has just recently become a topic in the research due to the internalization and growing number of variants in the automotive domain. Therefore, little research can be found in this area. We first begin with approaches which are based on the product line.

In [45], BMW Car IT has very roughly introduced their ideas of model-based HMI testing. Figure 3.3 is an original figure of this work which shows the work flow of their proposed approach:

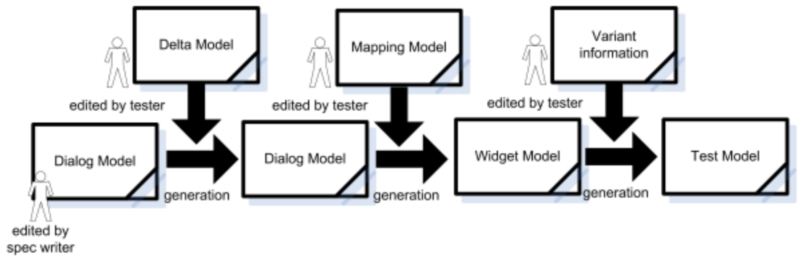


Figure 3.3.: Work flow of the specification based testing (original figure from [45])

The *dialog model* describes the GUI structure (contained widgets), the dynamic behavior and interfaces to underlying applications for all variants of the product line based on a domain-specific language. Simple variables are used to indicate for which variant a current GUI element should be displayed or a current interface is valid. *Widget models* are used to describe the behaviors of widgets. Since for different variants, different widgets are in use, widget models describe the product line specific behavior. A *mapping model* is used to map entities from the dialog model to their implementations in the widget model. In order to generate tests for a particular variant, a *variant specific test model* must be created. First, the dialog model is transformed into a *product line specific dialog model* based on a model transformation technique. Second, the product line specific dialog model is transformed into an *executable widget model* by applying product line specific mapping rules which are also not introduced. Finally, *variant information* is injected into the executable widget model to obtain the variant specific test model. Test generation is not addressed. Continuations of this work are not available. Due to the lack of technical details, it is quite difficult to compare the methods used. Nonetheless, the following points can be determined: first, in the specification language used, the dynamic behavior and the GUI structure are described in the same model and are strongly dependent on each other while a separation is more efficient for testing, as explained above. The merging of GUI structure and the dynamic behavior descriptions is also inflexible and work-intensive in the event of changes. Moreover, it is not possible to specify those cases in which variability exists in the menu behavior but not in the representation. Second, the work gives the impression that the problem of variability has not been sufficiently analyzed. HMIs in real life have very complex variability (Section 7.1 and 9.1.2) which is not manageable with single variables. Finally, the large challenge to avoid redundancies for the test generation and test execution has not been focused upon.

“AutomotiveHMI” [URLd] is a collaborative research project in which many leading German manufacturers and suppliers are participants and the author was also partially involved. The goal of the project is to optimize current infotainment system development processes and methods. With the participation of Audi, model-based testing of HMIs, and in particular modeling and testing variability, have become topics and a work package of the project. At the time of composing this thesis, the testing goals have been defined; however, there are still no concrete results available.

Although the testing approach introduced in [88] is not based on a software product line, it does address the testing of multiple GUI variants. The tool focuses on applications whose different front-end variants share the same business logic. The authors have mentioned Adobe Acrobat

Reader as an example of an application which runs on MS Windows, Mac OS, Linux, etc. The business logic is separated from the representation logic. The business logic is modeled with the transitional model-based testing tool Spec Explorer while the representation logic is recorded by a capture/reply tool. The testing tool provides a *GUI Test Generator* (GTG) which allows testers to define mappings between the business logic and the different variants of the representation logic. Based on these mappings, the "GTG converts business logic test cases into presentation logic test cases" for different variants. In the applications which are the focus of that work, the same business logic is bound to different front-ends which are separately captured and described. In comparison, the variability in infotainment systems' HMIs is much more complex. It exists both in the representation and in the business logic (menu behavior). In the representation, different variants share a huge set of commonalities. It is impractical to describe these commonalities repeatedly for each variant. Therefore, a solution must be found in order to specify the commonalities and the variations without redundancies for all variants. The largest challenge - to model and test variability in the menu behavior - does not exist in testing applications which have only one business logic.

3.3. Model-based development of GUIs with variability

Several works can be found relating to the domain of model-based development of GUIs with variability (area 3 in Figure 3.1).

In [16], a method is proposed for the model-based development of automotive HMIs based on the product line approach. In this approach, the menu flow of the HMI is described within the so-called *screen flow model*. The screen flow model is a *product line screen flow model* if it implicitly describes all possible variants and contains *variation points* indicating the boundaries between different variants. The *product line feature model* describes the features of the product line. A mapping between it and the product line screen flow model must be performed based on a mapping table. From the product line feature model, an *application feature model* can be derived which describes the features provided by a particular variant of the product line. Based on that and the mapping, a *variant screen flow model* can be generated from the product line screen flow model. This variant screen flow model describes the screen flow of the defined variant and is applied for the code generation. The product line feature model and variant feature model used are introduced in detail.

The commonality between that approach and ours is that both approaches face the problem of modeling the product line and modeling variability in the menu behavior. The ideas are quite similar: feature models are used to model the product line and variation points are used in the menu behavior model in order to indicate for which variants that particular menu behavior is valid. However, variability in the representation is not considered in that work. Another difference between the approaches is in the procedure for code/test generation. In the approach proposed in [16], for each variant the respective menu behavior is extracted from the menu behavior describing all variants of the product line. Automatic code generation is performed from the extracted behavior. This makes sense for the software development, since for different variants potentially different libraries and frameworks must be bound during code generation. In contrast, for model-based testing, our approach tries to avoid generating tests for each variant, since different variants share a large set of commonalities in the menu behavior and repeated test generation for each variant means that the common behavior is unnecessarily visited as often as the number of variants to be tested. Due to the usually large number of variants in practice, avoiding redundant test generations is an important goal in our approach. Additionally, in model-based GUI testing, a problem must be faced that does not exist in GUI development: generated tests for different variants contain a large set of identical tests which verify the common behavior. The test execution is very time-consuming, therefore the redundant execution of identical tests must be avoided. Finally, in practice, infotainment system HMIs do not only have features related to functions such as navigation and telephone, but also features related to the market and the language. The same function can be provided with different options. This complexity is not considered in the approach proposed in [16].

3.4. Conclusion

One major finding of the categorization and evaluation of related work is that for each related research area, several and often many approaches already exist. However, most of these approaches address only one or some of the sub-problems in comparison to this thesis. For instance, some address only the modeling of the GUI and some address only the testing of user action sequences.

Furthermore, the lack of industrial context leads to another problem: very few of the existing approaches face real-life complexity or aim for practical application. For instance, some approaches are based on specification languages which are not scalable for complex systems and some are based on very simplified product lines.

Finally, HMI testing under variability is still in its infancy. The characteristics of infotainment system HMIs and HMI product lines have not been sufficiently studied. Also, testing approaches are lacking which are based on real-life HMI development processes.

Part II.

Framework for Model-Based HMI Testing

As already explained, an HMI testing framework is still lacking which supports HMI integration tests, allows users to create an HMI test model describing the varied features of the HMI, supports automatic test generation and hence enables systematic test coverage.

In this chapter, our model-based testing framework for infotainment system HMIs is introduced which fulfills the above requirements. In this introduction, variability of HMIs is not yet considered. The framework forms the basis of this thesis and also the basis of the solution for testing under variability. The extension of the framework for testing under variability will be described later, in Part III.

Firstly, the essential components of the framework are introduced: the test-oriented HMI specification and the test generation. It will be discussed how extensions of additional components are possible in order to combine MBT with automated test executions. Afterward, possible HMI errors occurring in practice are identified. Based on these, we will clarify which kinds of errors can be addressed by the framework and which are within the focus of this work. Requirements for the test-oriented HMI specification will then be introduced in detail. This is also the main focal point of this part. Finally, a test generation algorithm and generated tests are introduced.

Chapter 4.

The Framework for Model-Based HMI Testing

This chapter first introduces the essential components of the proposed testing framework. Possible extensions for combining MBT with automated test executions are also discussed. Afterwards, possible HMI errors occurring in practice are identified. Based on these, we clarify which types of errors are within the focus of this work.

4.1. Components of the HMI testing framework

As introduced in Chapter 2.1, model-based testing (MBT) is a process in which test models are created and tests are automatically generated from these models. The MBT framework proposed in this thesis also contains these two essential components; however they are specialized for testing the infotainment system HMI. As presented in Figure 4.1, the test-oriented HMI specification acts as the test model and describes the expectation and features of the HMI which are to be tested. The test generation is able to generate tests from this test-oriented HMI specification.

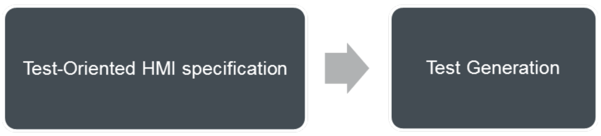


Figure 4.1.: Components of the MBT framework for infotainment system HMIs

Test-oriented HMI specification

The most important component in the framework is the test-oriented HMI specification, which specifies the expectation of the HMI and contains sufficient data to generate valid tests.

Depending on the individual HMI development processes of different manufacturers, a test-oriented specification can be achieved in different ways. For instance, it can be created by HMI specifiers who also have a testing background and therefore create a specification suited to the purpose of testing; however, in practice this is unusual. It is also possible for test engineers to work together with the specifiers on the same specification and extend it with test data. These methods correspond to the MBT variant presented in Figure 2.2. If the HMI specification is still completely informal, e.g. created as a PDF using Visio images, it is also possible to follow the MBT variant b) in 2.1 and create a separate test-oriented specification. However, as previously explained, this leads to more and possibly redundant work. It is also possible to follow the variant a) in Figure 2.1. As introduced in Section 2.2.3, if the development is carried out by the supplier, the development model can be extended into a test model. Our model-based testing framework does not require a particular MBT variant or a particular method to achieve the test-oriented HMI specification. Only the data defined in the requirements must be contained in the specification. This will be introduced in Chapter 5.

Test generation

In this thesis, we do not define any new test generation method or new coverage criteria. The test generation component in the HMI testing framework can be individually implemented based on different generation methods and adequate coverage criteria with the precondition that the generation method takes account of elements in the test-oriented HMI specification. For this component, we will introduce a test generation algorithm based on the transition coverage in order to illustrate the generation process and generated tests.

If generated HMI tests are to be automatically executed, the introduced framework must be extended with additional components supporting the test automation. Figure 4.2 shows the necessary extensions.

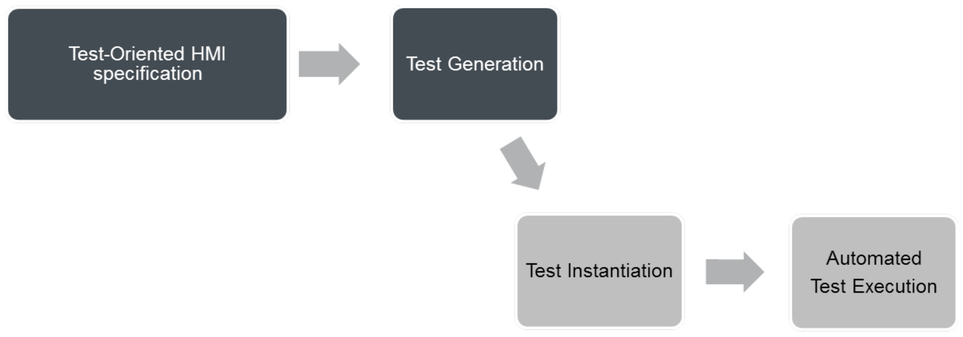


Figure 4.2.: Extended components of the framework supporting automated text execution

Test instantiation

As introduced in Section 2.1.2, generated tests contain abstract events and do not contain concrete user input data. These tests still cannot be automatically executed. Abstract events must be mapped onto test actions simulating the input of users or underlying applications. These test actions are usually created in the test automation framework and the framework can send them into the SUT in order to simulate user or application actions. In addition, concrete user data must be prepared in the test automation framework. Generated tests must be extended with this data.

Test instantiation is not a focal point of this work. The thesis [13] has introduced an instantiation method for feature interaction tests of infotainment systems.

Automated test execution

Test automation frameworks such as EXAM [64] and MODENA [URLh] are very popular in the automotive industry. These test automation frameworks allow testers to create tests, test expectations, executable test actions and any artifacts needed for automatic test executions. They also provide interfaces to the SUT, which is usually a test bench. Interfaces to the SUT are used to send stimulated user actions or ECU messages to the SUT. Interfaces from the SUT are also provided, via which the behavior of the SUT can be observed and then compared with the expectation. Currently, these test automation frameworks are mainly used for testing the functions as already introduced in 2.1.4. Figure 4.3 presents interfaces currently used in the MODENA framework at Audi for testing the infotainment system functions.

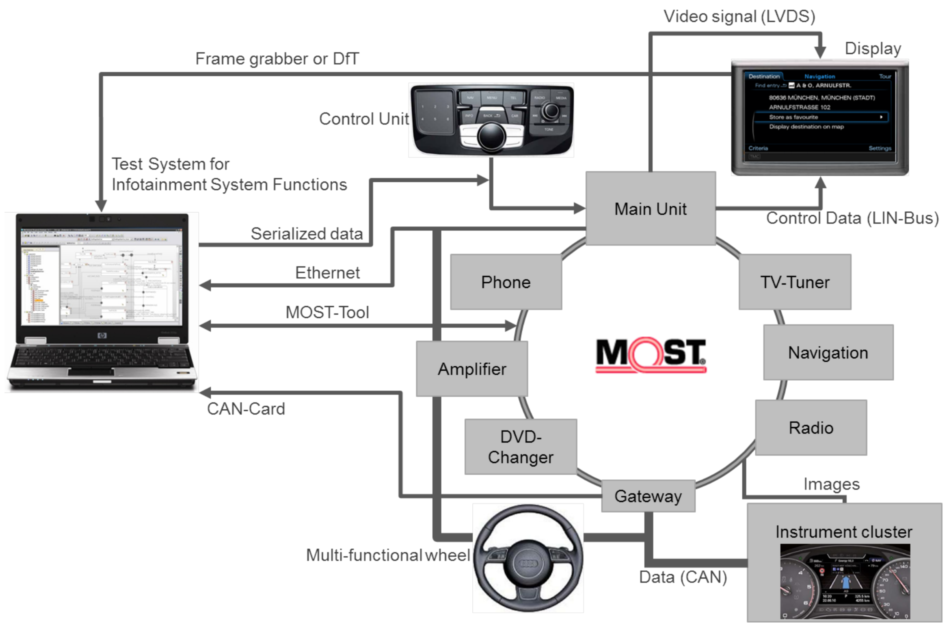


Figure 4.3.: MODENA test framework used for testing infotainment system functions

User actions such as pressing buttons can be sent in a serialized form to the main unit as if the user had physically pressed the buttons on the control unit. Via the interface to the MOST bus [47], bus messages such as “incoming call” can be simulated as if the messages had been sent by the ECU. Values and messages sent in the MOST can be captured and used to observe the SUT behavior. With the help of a frame grabber [57], screenshots can be captured from the currently displayed screen. With the help of image processing methods [18] such as icon and text recognition, the graphical and textual contents of the screen can be analyzed.

Such test automation frameworks can be used for the automatic execution of HMI tests. Currently available interfaces to the SUT are sufficient to execute generated HMI tests. In order to observe the dynamic behavior and the graphical representation of the HMI, frame grabbers or cameras can be used. Ideally, an interface can be provided which can report the name and possibly also the UI elements and their contents contained in the displayed screen.

4.1.1. Related work

In Section 3.1, a number of related works have already been introduced that propose frameworks for model-based GUI testing. Although the model types, specification languages and coverage criteria used are very different, all frameworks are composed of the same essential components: the test model describing the expectations of the GUI implementation and the test generator which generates tests from the test model. In the testing framework introduced in [13], test instantiation and automatic test execution are also considered. In [84], the framework additionally contains a regression tester.

4.2. Analysis of HMI error types

HMI errors are much more varied than function errors. In order to define which HMI errors the testing framework can and shall face, we must firstly find out which HMI errors can occur in practice. To do this, we have evaluated more than 200 HMI errors from a current practical HMI project. For the evaluation, an error is regarded as an HMI error if it is directly observable in the HMI. These errors are chosen from different phases of the project. We have classified the errors according to their error sources.

4.2.1. Errors occurring in practice

4.2.1.1 Menu behavior errors

In this work, failures caused by erroneous implementation of the menu behavior are called menu behavior errors. The symptom of this error type is either switching to an unexpected screen or missing menu changes. For example, one very frequent menu behavior error is that the user is led to an unexpected screen after he has pressed the return button. The expected behavior is either that the return button should lead to the screen that the user has just visited, or to the screen above the current one in the hierarchy. Developers often mistake these two concepts. Another frequent menu behavior error is an inconsistency in the menu navigation. In practice, the HMI implementation of a new generation is usually based on the old implementation.

Often, changes must be performed in many places e.g. due to a new menu navigation concept. Developers often fail to make the changes in all the necessary places. If the menu behavior is developed based on state diagrams, errors are commonly caused by the erroneous next state of transitions, wrong events, wrong conditions or wrong positions of history states.

In rare cases, the same symptom (wrong menu switch symptom) is not caused by erroneous implementation of the menu behavior, but by errors in the underlying applications. For instance, the menu behavior is correctly implemented as specified and two transitions starting from the same state are labeled with "if" and "else" conditions respectively. The values of these conditions are expected to be set by an underlying application during runtime. However, an erroneous assignment by the underlying application gives the impression that the menu behavior contains a failure. As explained previously, errors are classified based on the error sources. Therefore these types of errors are classified into the "application and framework errors" category.

4.2.1.2 Errors occurring inside screens

Errors of this class are observable in single screens and are caused by the erroneous implementation of single screens. They can be screen content errors, design errors and widget errors.

Screen content errors

Screens usually contain texts, icons and pictures. The statistical analysis has shown the following types of screen content errors:

Text content errors: A text field can present static or dynamic text. Static text is defined in the HMI specification or external text tools, e.g. the text title "navigation" in the screen 'NavAdrInput' presented in Figure 2.6. Wrong content in such static text is very common. For instance, instead of "navig." the text is defined as "navigation". Wrong line breaks in a long text can also be classified as text content errors.

A text field can also present dynamic text, which is not specified in the HMI specification or external text tools. Dynamic text content is usually sent from the underlying applications during runtime; for example, an incoming call number. In the evaluation, we encountered an error in which the HMI displays a set of zeros instead of the number for an incoming call. Experiences have shown that these types of errors are usually caused by underlying applications or interfaces between the HMI and underlying applications. Therefore they belong to the"application and framework errors" class.

Text order errors: Many screens contain more than one text row, the order of which is predefined in the specification. For instance, in the screen presented in Figure 2.5 the text "country" should be intuitively positioned above the text "city" and "city" should be positioned above "street". In practice, incorrect text order often occurs.

Language errors: In the early phases of an HMI development, language errors are especially frequent. A language error is, for example, where some German text still exists in a Chinese system due to the translation being overlooked. The distinction between language errors and text content errors is sometimes unclear; however, text errors caused by erroneous or lack of translations can be classified as language errors.

Icon or picture errors: The use of wrong icons also occurs in practice. Usually, the HMI implementation of the new generation is based on the old implementation. In the early phases of development, the replacement of old icons with new ones is frequently overlooked.

Icon order errors: Similarly to the texts, icons should also be presented in a predefined order. For instance, in the status bar shown in Figure 2.6, icons representing different statuses such as the current time, traffic information, incoming messages or signal strength have a predefined order. Although these icons are not all displayed at the same time, the order must be maintained. The evaluation has shown that errors involving wrong icon orders have occurred.

Missing or unexpected UI elements: Texts or icons can also be completely missing or occur unexpectedly in a screen. A very frequent error is that an empty row is missing e.g. between an information text field and the confirmation button. Some widgets are not constantly displayed in the screen but only under particular runtime conditions, for example the icon indicating an incoming call. An error in which such a widget is completely omitted from the screen belongs to this current class. If it is correctly displayed in the screen, but does not appear when the runtime condition is fulfilled, then this error is caused by erroneous implementation of the widget behavior and belongs to the widget behavior error class.

Missing or unexpected elements can also be caused by underlying applications. For example, in the case of an incoming message, the underlying application must send an event to the HMI, which afterwards displays the 'incoming message' icon in the screen. However, if the underlying application sends no event or the wrong event e.g. the event for an incoming call, an error is observable in the HMI, but is not caused by the HMI. These types of errors are also classified as "application and framework errors".

Wrong types of widgets: In some cases, the wrong type of widget is used in the screen, e.g. a check box is used instead of a drop-down list.

Design errors

Design rules define how screen contents should be represented in terms of position, distances, colors, fonts etc. Errors caused by breaking the design rules are design errors. Shifting of pixels is a frequent design error. Design rules also define how a long text displayed in one row should be handled. For instance, up to a certain number of characters may be shown and the remaining characters replaced with "...". We came across several errors in which the text was not correctly cut.

Widget behavior errors

Infotainment system HMIs usually contain many intelligent widgets with their own behavior, e.g. the scroll list in Figure 2.6. Errors occur frequently in widget behavior; for instance, the scroll list does not display the arrows indicating previous or next pages. The behavior of a drop-down list or check box can also be erroneous. For instance, a drop-down list should offer the"off" option when the bluetooth interface is on, but continues to display the "on" option. As explained previously, a simple icon can also have a particular behavior; for example the incoming call icon. In an erroneous case, the icon does not appear or disappear. Many texts and icons demonstrate the behavior of being either active or deactivated. In Audi's HMI, inactive elements are displayed in the color gray. The evaluation has shown that many errors occur in which elements are not inactivated even though the options provided by these elements are not available.

4.2.1.3 Pop-up errors

Infotainment system HMIs usually contain a large number of pop-up windows which are used to provide additional information and are either invoked by the user or initialized by underlying applications For instance, as a reaction to a change in the sound volume which is triggered by the user, a partial pop-up window is displayed in order to show the scalar. Another example is the pop-up that is displayed as a reaction to an event sent by underlying applications, e.g. “empty tank”. Errors can occur in connection with pop-ups, e.g., erroneous contents and wrong priorities of different pop-ups.

Sometimes, errors detected in connection with pop-up windows are caused by underlying applications, overloaded bus systems, or the board computer. For example, the underlying application sends a wrong message to the HMI, so that the HMI reacts with a different pop-up than expected. These types of errors are classified as “application and framework errors”.

4.2.1.4 Application and framework errors

In many of the cases above, it was explained that some of the errors observable in the HMI are not caused by erroneous HMI implementation, but by underlying applications, bus communication or any other components in the HMI environment. In very few cases, errors can also be caused by an erroneous HMI framework, e.g. bugs in the event queue.

Figure 4.4 presents the error distribution based on the classification introduced above. This statistical analysis should give an indication of the HMI error distribution in practice. The distribution can be very different if other projects, development phases, error tickets and classification criteria are chosen. For example, Daimler’s statistical analysis is based on the symptoms of the errors. In their analysis results, there is a greater percentage of pop-up errors, and a similar percentage of menu behavior errors.

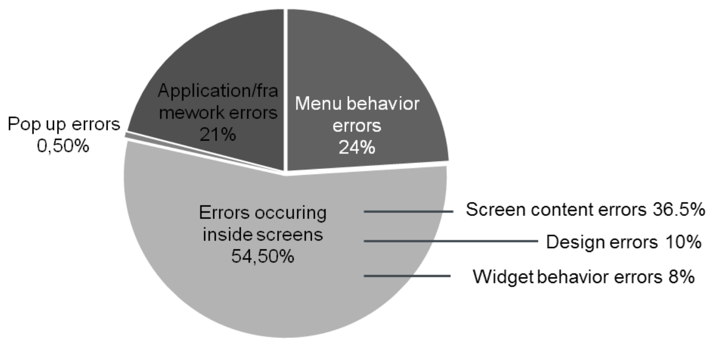


Figure 4.4.: Distribution of errors observable in the HMI

Figure 4.5 presents the distribution of true HMI errors:

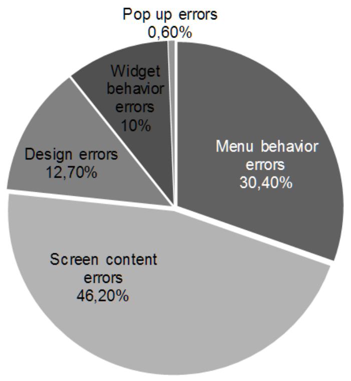


Figure 4.5.: HMI error distribution

This chart clearly shows that menu behavior errors and screen content errors make up the greater part of HMI errors. There is a high percentage of menu behavior errors, but they have a single error source, namely the behavior model in a model-based development. This means that if a specification of expected menu behavior is available and automatic test generation is applied, about 30% of HMI errors can be detected, according to the claims of MBT approaches that test coverage can be systematically achieved. The specification of the expected screen contents also requires more effort, which is currently low in relation to the high percentage (about 46%) of the HMI errors that are screen content errors. This means that if an HMI testing framework focuses on menu behavior tests and screen content tests, a significant effect can be achieved with low efforts.

4.2.2. HMI errors addressed in this thesis

In the last section, the types of HMI errors that occur in practice were introduced. Of these, menu behavior errors and screen content errors, which make up the greater part of HMI errors occurring in practice, are the focal points of this thesis.

Detecting design errors is another research area. Currently, a thesis focusing on design tests of infotainment system HMIs is in process [50]. In order to test the behavior of intelligent widgets, their behavior must be specified in a similar way to the menu behavior. Widget behavior tests are not a focal point of this work. Both design tests and widget behavior tests must also be driven by menu behavior tests. Pop-ups usually have their own behavior such as to appear and disappear. This behavior can be described similarly to the menu behavior. A characteristic of pop-ups is the priority between different pop-up menus. If pop-up tests are intended, the behavior and the priority should be the test focuses. Pop-up tests are not within the focus of this work. The logic of underlying applications and the interactions between different components in the HMI environment are not describable in an HMI specification. Also, in the current HMI development process in practice, there can be no complete specification describing all sub-components of the infotainment system and their interactions. Specifications are only available for each component and the interfaces between them, because such a huge infotainment system is developed by a number of suppliers. Therefore, application and framework errors cannot directly be detected by testing the HMI and are not within the focus of this thesis.

Menu behavior errors

Detecting menu behavior errors is the main focal point of this work. In this thesis, tests which are created for detecting menu behavior errors are called menu behavior tests and can be formally defined as follows:

Definition. *A **menu behavior test** is defined as a tuple MTest = (Cons, Steps) where Cons is a set of conditions and Steps is a sequence of test steps. If Cons is empty it must not be explicitly declared. The sequence Steps is defined as Steps =* $\langle$ *step*$_0$*, ..., step*$_k$ $\rangle$ *where step*$_i$ *with* $0 \leq i \leq k \in AC \cup EXP$*. AC is a set of test actions which are abstract events generated from the behavior model of the test-oriented HMI specification. EXP is a set of expectations in the form of names of expected screens. step*$_0$ *must be the name of the initial screen, which is expected if the SUT is started.*

A menu behavior test shall only be started if all conditions $\in$ Cons are fulfilled.

The following is the menu behavior test verifying the menu behavior described in Figure 2.7:

(NavAdrInput) $\rightarrow$ (country) $\rightarrow$ [NavCountryList] $\rightarrow$ (chooseCountry) $\rightarrow$ (NavAdrInput) $\rightarrow$ (city) $\rightarrow$ [NavCitySpeller] $\rightarrow$ (enterCity) $\rightarrow$ (NavAdrInput) $\rightarrow$ (street) $\rightarrow$ [NavStreetSpeller] $\rightarrow$ (enterStreet) $\rightarrow$ (NavAdrInput) $\rightarrow$ (start) $\rightarrow$ [NavCalculation] $\rightarrow$ [NavMap]

In this sample test, the condition set Cons is empty. The set Steps is composed of both test actions and expectations of screens from which the expectation is shown in "[ ]" and the test actions are shown in "( )". In this thesis, we will continue to use this notation for menu behavior tests. Later, it will be shown that these test actions are events in the test-oriented HMI specification which trigger transitions.

These generated tests must be initialized for automatic test executions. Abstract events such as 'country' must be mapped to test actions simulating the user's focusing on the first line and pressing enter on the control unit. Events such as 'chooseCountry' must be mapped to a test action which is bound to concrete user input data such as "Italy". During the automatic test execution, in order to verify whether an expected screen is presented on the display, either an interface that exports the screen name or some screen recognition method [10] must be available.

Menu behavior tests mainly detect menu behavior errors. They also coincidentally detect some of the application and framework errors such as a delay in screen change caused by overloaded bus systems or wrong events sent by underlying applications. The precondition of generating menu behavior tests is that the expected menu behavior is specified in the test-oriented specification.

Screen content errors

Screen content tests are also a focal point of this work. Firstly, the definition of screen content tests is introduced.

Definition. *Tests which are used to detect screen content errors or a subset of screen content errors are called **screen content tests**.*

The following shows how the menu behavior shown above can be extended with a screen content test:

[NavAdrInput] → {*navadrinput.title.text, mask_title*} →
{*navadrinput.subtitle.text, mask_subtitle*} → (nav) ...

Steps in "{ }" are extended test steps, which describe the expectation of screen contents. Step {navadrinput.title.text, mask_title} is composed of a text label ID "navadrinput.title.text" and a mask ID "mask_title". This means that the text content saved under the text label ID is expected in the pixel area defined under the mask ID. Later, in Chapter 5.1, it will be explained that due to the variety of languages provided, text contents should not be directly specified as properties of widgets, but externally in the so-called text tool. A text widget only has a text label ID as a property which is a reference to the text content. Similarly, it will be explained that design information such as the position of a text, which is defined as a mask (Section 5.1), should also be specified externally. A widget contains only a mask ID as a property. Therefore, generated screen content tests still only contain text label IDs and mask IDs. For automated test execution, this screen content test also has to be initialized. The activated language of the current SUT to be tested is usually known and is available as a global variable in the test automation framework. The text content in the correct language and the coordinate of the mask must be obtained. An initialized screen content test contains the following data for the English language:

[NavAdrInput] → {*"Navigation", "offsetX=241;offsetY=5;sizeX=318;sizeY=55"*} →
{*"Address", "offsetX=30;offsetY=60;sizeX=750;sizeY=45"*} → (nav) ...

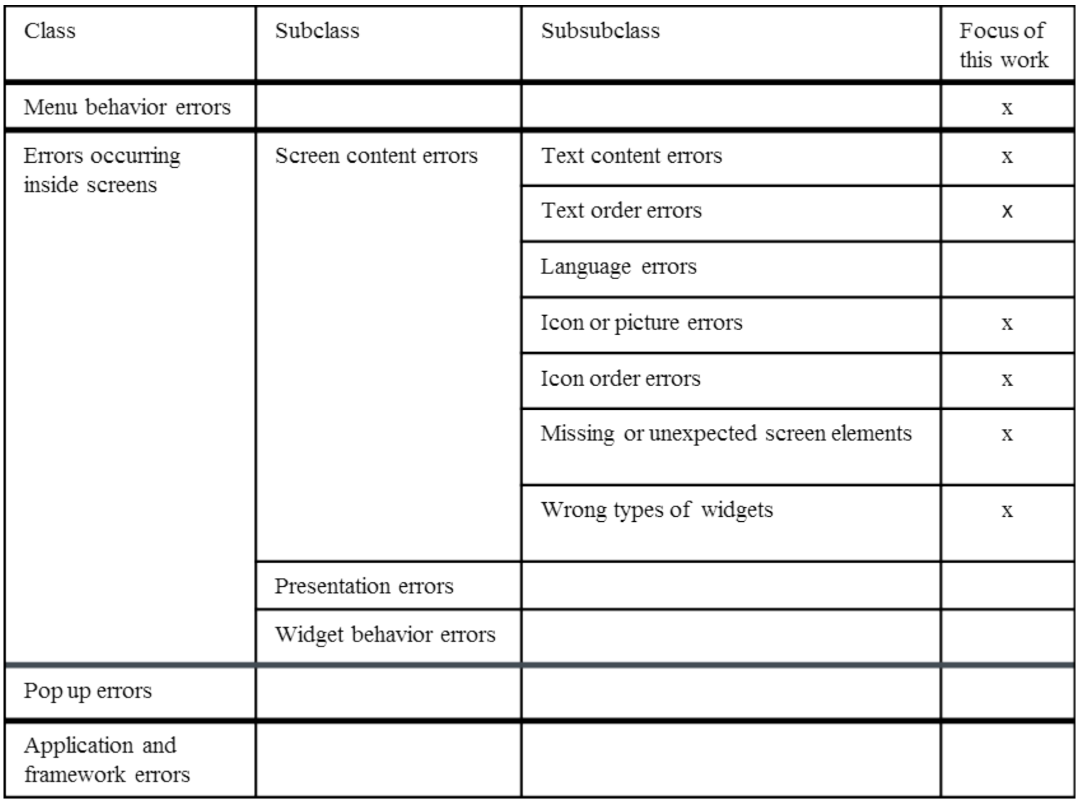
Screen content tests are driven by menu behavior tests. This means the screen content tests cannot be performed until a menu behavior test has driven the SUT to the particular screens whose contents are to be tested.

With screen content tests as demonstrated above, text contents and text orders can be verified. Missing texts, too, can be detected. Testing the icons and widget types is very similar. For an icon or a widget, the reference to the icon or widget template must be given as a property instead of the text label ID. Errors can be detected with the help of picture recognition using image processing. Language errors are not directly detectable. Tools such as language checkers must be used and native speakers are usually required. However, menu behavior tests can be used to support the detection of languages errors. During the execution of menu behavior tests for foreign language systems, screenshots of the visited screens can be captured. Native speakers do not have to manually execute these test themselves. They can easily review the captured screenshots on a PC. Testing language is not within the focus of this thesis and will not be further discussed at this point.

4.3. Conclusions

As introduced in Chapter 2.1.4 and Chapter 3, current model-based testing frameworks in practice and in research literature mostly address only function tests or tests for standard PC applications. In order to perform model-based testing for infotainment system HMIs, a testing framework specialized for HMI tests has been introduced in this chapter.

Essential components of the framework were introduced. The test-oriented HMI specification acts as the test model. It describes the expectation of the HMI and contains sufficient data for the test generation. Requirements of the test-oriented HMI specification will be introduced in the next chapter. The test generation component is able to generate HMI tests from the test-oriented HMI specification and will also be introduced in a later chapter. It was discussed how the framework can be extended for automated test execution. Afterwards, the HMI errors that occur in practice were introduced. Based on these, we clarified the error types to be addressed in this work. Figure 4.6 provides an overview of the HMI errors to be addressed:



Class	Subclass	Subsubclass	Focus of this work
Menu behavior errors			x
Errors occurring inside screens	Screen content errors	Text content errors	x
		Text order errors	x
		Language errors	
		Icon or picture errors	x
		Icon order errors	x
		Missing or unexpected screen elements	x
		Wrong types of widgets	x
	Presentation errors		
	Widget behavior errors		
Pop up errors			
Application and framework errors			

Figure 4.6.: HMI errors addressed in this work

Chapter 5.

The Test-Oriented HMI Specification

In the last chapter it was stated that the main focal point of this thesis is testing the menu behavior. Screen content tests should also be addressed. The precondition to do these tests is that the expectations of the menu behavior and the screen contents which are to be tested must be specified. In this thesis, they must be specified in the so-called test-oriented HMI specification. In particular, the specification must contain sufficient data to generate valid tests. In this chapter, the requirements of the test-oriented HMI specification will be introduced.

5.1. General requirements

Informal specifications consisting of, for example, PDF, Visio and Photoshop files cannot be used for test generation. In order to generate tests automatically, the test-oriented HMI specification must fulfill the following requirements:

***Requirement 1**: The notation, syntax and semantics which are used in the test-oriented HMI specification must be precisely defined.*

In this thesis, we do not define requirements for which specification languages should be used for the test-oriented HMI specification. The requirement is that for each specification language used, the notations, syntax and their semantics must be precisely defined. This is necessary for the test generation. For instance, UML SC is a very popular specification language for describing dynamic behavior which, however, leaves the notation and semantics open. In order to apply a code generator to automatically derive software code from a model which is described with UML SC, the notation and syntax used, and their semantics, must be precisely defined and taken into account by the code generator.

In order to specify the screens for screen content tests, it is sufficient to describe the screens abstractly. Development tools such as EB GUIDE Studio and VAPS XT allow users to create concrete screens with available widget libraries or to integrate self-created widget libraries. For a specification, concrete screens are not necessary and lead to unnecessary complexity. There are many specification languages that can be used for an abstract description of screens such as OEM XML [7], UIML [99], XUL [32] and XAML [73] which will be introduced in Section 5.4.

No matter which specification languages are used for the test-oriented HMI specification, their semantics must be well-defined. Furthermore, the required data (introduced in later chapters) must be specified.

Separation of different aspects, as the model-view-controller pattern [30] proposes, has advantages such as interchangeability and reusability. The separation is also relevant for the test-oriented HMI specification. The specifications of different HMI contents and also the generation of different types of tests should be independent of each other. For instance, if screen content tests are not intended, the menu behavior should be described without the specification of the screen contents. Additionally, if a design rule or some texts are changed, changes should not be necessary in the menu behavior specification and the screen specification. Furthermore, different teams are responsible for different parts of the HMI specification. For instance, designers are responsible for the design rules and the text writers and translators are responsible for the texts. Our test-oriented HMI specification must also allow separate specifications for different features of the HMI. The outcome of this is the following requirement, which has already been introduced in a previous work [29]:

Requirement 2: *The test-oriented HMI specification should have a layered structure and the following layers: the behavior layer, the data and the event layers, the representation layer, the design layer and the text layer. Figure 5.1 presents these layers.*

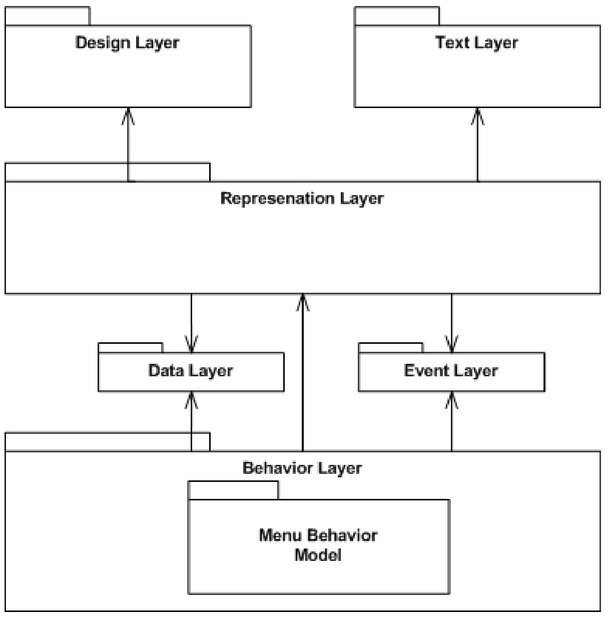


Figure 5.1.: Layers of test-oriented HMI specification

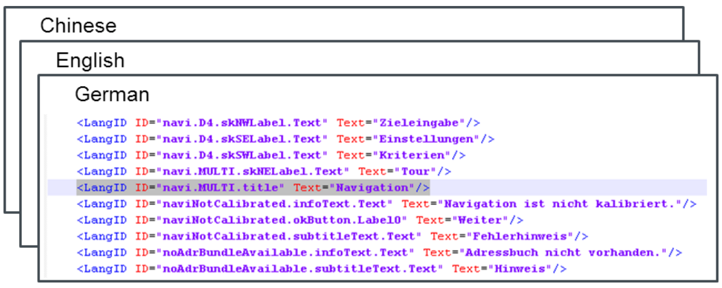
Data and event layer Variables and events needed for the specifications in other layers can be defined in the data layer and event layer.

Behavior layer Dynamic behaviors such as the menu behavior, widget behaviors and pop-up behaviors should be specified as separate models in the behavior layer. The behavior layer contains at least a menu behavior model and is mandatory, since all other HMI tests must be driven by menu behavior tests. As introduced in 4.2.2, since the widget behavior and pop-ups are not addressed in this thesis, we focus only on the menu behavior model in the behavior layer. The menu behavior of the HMI should be specified as a **menu behavior model**. The menu behavior model must contain sufficient data for generating menu behavior tests. Requirements for the data will be introduced in Section 5.2.

Representation layer In the representation layer, the representations of screens are specified. The specification includes the UI elements contained in the screen such as a title widget and the properties such as the position of the title widget. If the test-oriented HMI specification is used both as the specification and the test model, as in the MBT variant shown in Figure 2.2, all screens and all contained widgets of the HMI should be specified. The set of specified screens and widgets which are to be tested must be indicated. If the test-oriented HMI specification is only used as a test model, then only the screens, widgets and properties which are to be tested need to be specified in the representation layer. Requirements for the representation layer will be introduced in detail in Section 5.3.

As explained in Section 4.2.2, the texts and the design information should not be directly specified in the representation layer. Therefore, the representation layer has access to the design layer and text layer as shown in Figure 5.1. The relationship from the behavior layer to the representation layer is unidirectional. This will be introduced in Section 5.2.

Text layer If verification of texts is intended, the expected texts must be specified in the text layer. The text layer is therefore optional. As already introduced in Section 4.2.2, the HMI is a very international product and is usually offered in many different languages. It is inefficient and even impossible to specify texts in all available languages directly as properties of screens in the representation layer. A separated text layer should also simplify the translation work of different native speakers. Each text should be specified with a unique text label. Texts in different languages should be specified in different lists or files. Figure 5.2 demonstrates an example:

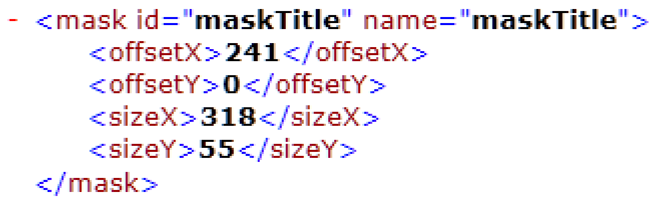


```
<LangID ID="navi.D4.skNWLabel.Text" Text="Zieleingabe"/>
<LangID ID="navi.D4.skSELabel.Text" Text="Einstellungen"/>
<LangID ID="navi.D4.skSWLabel.Text" Text="Kriterien"/>
<LangID ID="navi.MULTI.skNELabel.Text" Text="Tour"/>
<LangID ID="navi.MULTI.title" Text="Navigation"/>
<LangID ID="naviNotCalibrated.infoText.Text" Text="Navigation ist nicht kalibriert."/>
<LangID ID="naviNotCalibrated.okButton.Label0" Text="Weiter"/>
<LangID ID="naviNotCalibrated.subtitleText.Text" Text="Fehlerhinweis"/>
<LangID ID="noAdrBundleAvailable.infoText.Text" Text="Adressbuch nicht vorhanden."/>
<LangID ID="noAdrBundleAvailable.subtitleText.Text" Text="Hinweis"/>
```

Figure 5.2.: Text Layer

Design layer If screen content tests are intended, the required design information should be specified in the design layer. Design information includes positions, colors, distances, fonts, and rules such as where a long text should be cut. A UI element specified in the representation layer must be represented according to the design rules specified in the design layer. UI elements in the representation layer should only contain properties referencing the design rule it should follow. Which design information should be specified depends on the testing goals. For instance, if texts should be tested, the pixel areas in which the texts are expected must be given. If icons should be tested, the reference pictures of these icons and their pixel areas must be specified.

A pixel area in which a UI element is expected should be specified as a **mask**. Figure 5.3 presents a mask defined for the title widget:



```
- <mask id="maskTitle" name="maskTitle">
      <offsetX>241</offsetX>
      <offsetY>0</offsetY>
      <sizeX>318</sizeX>
      <sizeY>55</sizeY>
  </mask>
```

Figure 5.3.: Mask definition for the title widget

5.2. Requirements for the menu behavior model

In this section, we focus on the problem of how to specify the menu behavior and which data must be specified so that the menu behavior model is ready for the generation of menu behavior tests.

As stated in Section 5.1, we do not define requirements regarding the specification language used for describing the menu behavior. Our requirements are confined to the data which must be specified in the menu behavior model. Our proof-of-concept models are based on UML SC, since in practice UML SC is very widely used in HMI development and a variety of tools are available. Also the introduction of the proposed approaches and examples in this thesis is based on the assumption that the menu behavior should be specified with UML SC.

We have evaluated a small formal behavior model describing a function and discovered that such a model is not adequate to describe the menu behavior of the HMI. We also evaluated the menu behavior SC of a current development model (Section 2.2.3) for test generation and discovered that although this menu behavior model is formal and created for describing HMI menu behavior, some of the data which is required to ensure the generation of valid tests is lacking. This data must be available in the menu behavior model of the test-oriented specification. In the following section, we will introduce those situations in which additional data is necessary and our solutions to provide this data.

View state

Following the assumption that the menu behavior should be described with UML SC, UML element state is used to represent a possible status of the menu behavior. As soon as the HMI has achieved some of the states, particular screens must be presented on the display. The connection from such states to the associated screens must be specified in the menu behavior model. We proposed to define a new state type **view state** to specify this connection as in several HMI development tools. In the following paragraphs, some basic definitions will be introduced:

Work [11] has introduced a syntax defining the semantics of UML SC. In [44], this syntax is used to define simple states, or-states and and-states. We reuse the same syntax and the definitions in [44] as a basis for the definition of view states:

Definition. *A **SC** is defined as a tuple $SC = (S, TR, E$ and A $(E \subseteq A))$ where S, TR, E and A are respectively a finite set of states, transitions, events and actions of the SC.*

Definition. *A **simple state** also called **basic state** is defined as $s \in S$ with $s = [n]$ where n is the name of s.*

Definition. *An **or-state**, also called **composite state** or **compound state**, is defined as $s = [n, (s_1, ..., s_k), T] \in S$ where $name(s) = n$ is the name of the state s. Here $s_1, ..., s_k$ are the simple states which are contained in s or any sub-or-states in s, also denoted by $sub\_state(s) = s_1, ..., s_k$. $initial(s) = s_{init}$ is the initial state of s. $T \subseteq TR$ is the set of internal transitions of s.*

We define a view state as:

Definition. *A **view state** is a simple state $s = [n, vn]$ where $name(s) = n$ is the name of the state s and $view\_name(s) = vn$ is the name of the associated screen.*

In Section 5.3, it will be introduced that a view state in the test-oriented HMI specification is actually associated with an abstract screen, i.e. an abstract description of a screen. As explained in Section 5.1, in a specification no concrete screens need be developed.

In many HMI development tools, a graphical notation is used for a view state as shown in Figure 5.4. The view state is presented with the associated concrete screen in it.

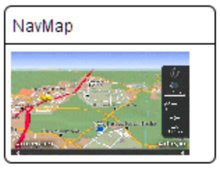


Figure 5.4.: Graphical notation for view states in many HMI development tools

In the test-oriented HMI specification, since screens are only abstractly described, we use the following graphical notation:

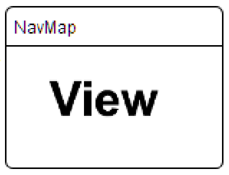


Figure 5.5.: Graphical notation for view states in the test-oriented HMI specification

In the menu behavior model, more than one view state can be associated with the same screen, but a view state cannot be associated with more than one screen. Intuitively, the HMI switches from one screen to another. Therefore, in many menu behavior models, all simple states are view states. Developers of these menu behavior models tend to follow their intuition and do not use simple states to present some inter-states, in which no screen must be displayed. From this point onward, we will assume that all simple states used in the menu behavior model of the test-oriented HMI specification are view states.

Screen-dependent transition

A real-life infotainment system HMI usually has a very complex menu behavior and contains a large number of screens. Consequently, a large set of view states are required in order to describe the menu behavior. Most graphical modeling languages allow the use of diagrams and hierarchies, since it is impossible to describe the complete menu behavior within one diagram.

This, however, leads to a problem: two sub-states which are contained in different diagrams cannot be directly connected to each other. For instance, the menu behaviors for the navigation function and the telephone function are described in two separate or-states ‘NavigationRoot’ and ‘TelephoneRoot’, the contents of which are described in different diagrams, as shown in Figure 5.6. A transition is needed to connect a sub-state in ‘NavigationRoot’ and a sub-state in ‘TelephoneRoot’ in order to allow the user to choose an available address saved in the phone book directly from the navigation function.

Creating a normal transition which directly connects the parent or-states within the same diagram, such as the transition labeled as the ‘confirm’ event in Figure 5.6, would describe the wrong behavior, since the semantics of a normal transition define that it connects each sub-state of the starting or-state with the initial state of the target or-state. A normal transition and its semantics can be defined as follows:

Definition. *A* ***transition*** *is defined as a tuple* $t = (n, s_{source}, e, c, \alpha, s_{target})$ *where* $name(t) = n$ *is the transition name,* $source(t) = s_{source} \in S$ *and* $target(t) = s_{target} \in S$ *are source and target states of* t*, respectively,* $ev(t) = e \in E$ *the trigger event,* $cond(t) = c$ *the trigger condition, and* $acc(t) = \alpha$ *is the sequence of actions that are carried out when a transition is triggered. The trigger condition is defined as* $c = [n, f]$ *where* n *is the name of the condition and* f *a function of type boolean.* [11]

The semantics of an outgoing transition $t = (n, s_{source}, e, c, a, s_{target})$ starting from an or-state $s_{source} = [n, (s_0, ..., s_k), l, T]$ and pointing to an or-state $s_{target} = [n', (s'_0, ..., s'_j), l', T']$ where

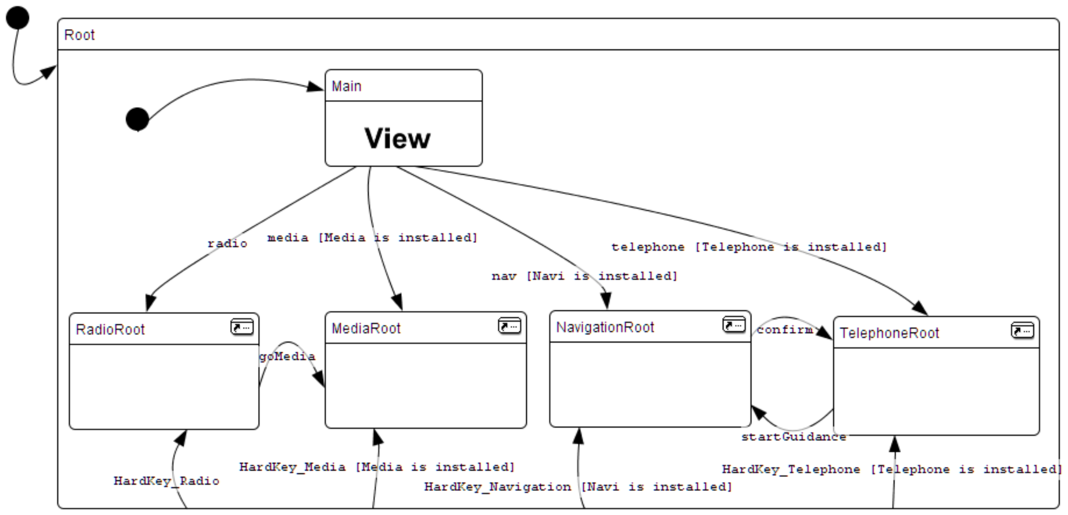

Root
Main
View
radio
media [Media is installed]
telephone [Telephone is installed]
nav [Navi is installed]
RadioRoot
MediaRoot
NavigationRoot
TelephoneRoot
goMedia
confirm
startGuidance
HardKey_Radio
HardKey_Media [Media is installed]
HardKey_Navigation [Navi is installed]
HardKey_Telephone [Telephone is installed]

Figure 5.6.: Screen-dependent transitions

s'_0 is the first simple state in s_{target}, is a set of transitions $t_0, ..., t_k$ where source$(t_i) = s_i$ and target$(t_i) = s'_0$ for $0 \geq i \geq k$.

To resolve this problem, a new type of transition is defined: the **screen-dependent transition**. It is defined as follows:

Definition. *A transition t = (n , s_{source}, e, c, a, s_{target}, VS) where s_{source} = [n,(s_0 ,...,s_k),l, T] is an or-state and s_{target} is a either simple state or also an or-state s_{target} = [n',(s'_0 ,...,s'_l),l', T'] is a* ***screen-dependent transition****, if t is only allowed from a subset of the sub-states of s_{source}: $VS \subseteq \{s_0, ..., s_k\}$ and allowed to point to a sub-state $s'_j \in \{s'_0, ..., s'_l\}$ of s_{target} which is not necessarily inital(s_{target}) in the case that s_{target} is an or-state.*

Such a transition is called a screen-dependent transition, since the set VS is dependent on the screens specified in the representation layer (Section 5.3); only view states which are associated with screens which can trigger the event e can be a member of VS. The set VS can be calculated based on the triggerable events of screens in the representation layer. If s'_j is not the initial state of s_{target}, then it must be explicitly defined.

The semantics of a screen-dependent transition $t = (n, s_{source}, e, c, a, s_{target}, VS)$ with $s_{source} = [n, (s_0, ..., s_k), l, T]$ and $s_{target} = [n', (s'_0, ..., s'_l), l', T']$, is a set of transitions $t_0, ..., t_n$ with source$(t_i) = s_p$ and target$(t_i) = s'_q$ ($1 \leq i \leq n$, $1 \leq p \leq k$ and $1 \leq q \leq l$) where event e is triggerable from the associated screens of s_p and t is allowed to point to s'_j. If the s_{target} is a simple state, target$(t_i) = s_{target}$.

Global transitions

In an infotainment system, there are user actions which are effective in any state and from any screen of the system. For instance, the buttons in the control unit (Figure 0.1), which are usually called hard keys, allow the user to directly access any function from any screen and at any time. Activities of this type can be described with the so-called global transitions:

Definition. *A* ***global transition*** *is a transition global_tr* $= (n, s_{source}, e, c, a, s_{target})$ *where* s_{source} *is the root state of the SC. The definitions of the remaining elements are identical as for a normal transition introduced above.*

The set of global transitions is defined as GlobalTR $\subset$ TR of the SC.

Figure 5.6 shows four global transitions allowing the user to directly access the four functional features. They start from the root state and point to the four respective functional feature states.

Sufficient conditions in the menu behavior model are necessary to ensure the generation of valid tests, correct test execution and the reporting of correct testing results. In this thesis, **PreStepsConditions**, **PrepareConditions** and **RuntimeConditions** are defined in order to specify the necessary conditions.

PreStepsCondition

In many situations, some user actions are allowed only if some other actions have been previously performed. For example, to define a navigation goal, a country must be chosen before a city can be entered, as shown in Figure 2.7. This means that before a country has been entered by the user, the condition for the transition triggering the city input is not yet fulfilled and hence it is not possible for the user to enter a city. In the evaluated HMI development model, this complexity is wholly implemented in the underlying application. The transition triggering the city input is labeled with a condition containing the string "[...]" as shown in Figure 5.7, whose variables can be set by the underlying application. The underlying application calculates whether a city input is allowed and informs the HMI of this by assigning the values of the condition. This type of implementation is very popular due to the principle of a light GUI which should not contain much complexity. However, this solution leads to a serious problem for test generation. Since this logic is irreproducible in the menu behavior model, many generated tests would contain an illegal order of test steps; for example, entering a city first and then a country. At the time of creating an HMI specification, interfaces to underlying applications are usually still undefined. Nevertheless, the behaviors of underlying applications are either not available or not available in a useful form for the test generation. Therefore, in order to avoid creating invalid tests, such logic must be completely specified in the test-oriented HMI specification.

There can be different ways to specify the allowed orders of user actions. One way is to define more view states associated with the same screen 'NavAdrInput' which represent different intermediate states; for example, view state A presents the status that no country has been entered yet and view state B presents the status that a country has been entered. However, this way is unintuitive since according to intuition each screen presents a unique state of the

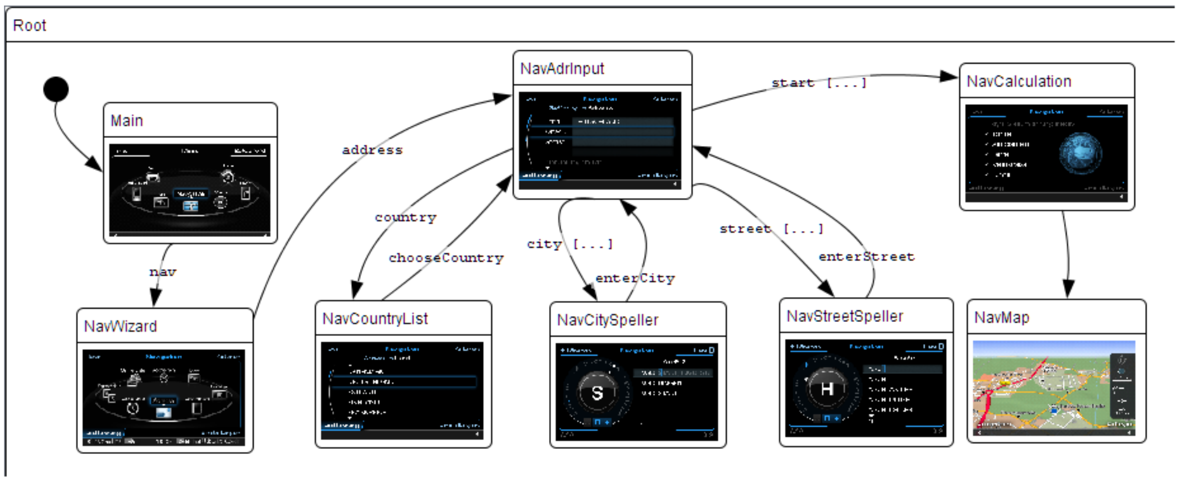


Figure 5.7.: A menu behavior model in which the allowed order of user actions is not directly specified

menu behavior. Furthermore, this solution would very quickly lead to a state explosion. Also, in practice, this solution is unusual in HMI specifications or development models.

We will solve this problem with a condition type PreStepsCondition, which can be defined as follows:

Definition. *A* ***PreStepsCondition*** $c = [n, f]$ *is a normal condition as previously defined which has a special feature: for each* c *there exists at least one transition* $t \in TR$ *labeled with at least one action* ac *which makes* $f = true$*.*

Figure 5.8 shows how PreStepsConditions are used in our proof-of-concept menu behavior model to define the step orders. The condition 'cityAllowed' of the transition triggering a city input is initially not fulfilled when the test generation is started. Only when the test generation has selected the transition labeled with the event 'country', is this condition fulfilled by the action 'allowCityStreet'. This means that the test generation calculates the conditions before it selects a transition and selects only those transitions whose conditions are fulfilled. In this way, correct orders of user actions are ensured and illegal tests can be avoided.

PreStepsConditions are evaluated during the test generation. They cannot be fulfilled before the test generation is started. Once the tests are generated, they are not relevant any more for the test execution.

PrepareCondition

In many cases, generated tests need preconditions before they can be executed. In comparison to the PreStepsConditions, the preconditions cannot be fulfilled by the tests themselves during the test execution, but must be fulfilled by test engineers before tests are started. For instance, the precondition of all navigation tests is that the navigation function is installed on the SUT and the correct navigation data base is connected.

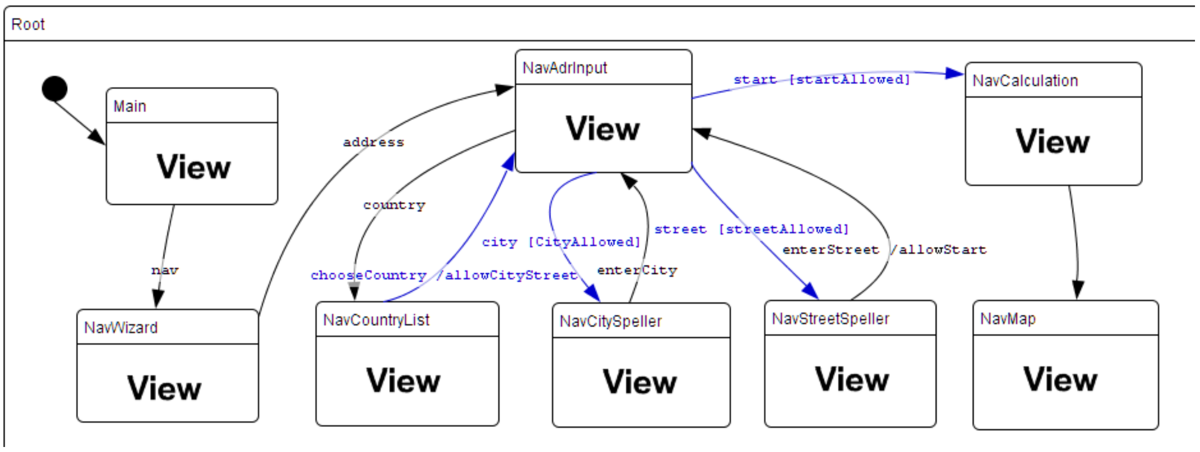


Figure 5.8.: PreStepsConditions

Preconditions of tests must be prepared by test engineers before the test execution, either manually or automatically. In order to advise the test engineers of the necessary preparation, the test generator should add such preconditions into the conditions sets Cons of the respective tests.

To specify such conditions, the type PrepareCondition is defined as follows:

Definition. *A* ***PrepareCondition*** *is a condition* $c = [n]$ *where* n *is a meaningful name which advises the test engineers of preparations they must make before they start the test.*

In the cases we encountered, a PrepareCondition does not have to contain a value specification since these conditions are never verified, but are usually used for preparation. It is possible to define some RuntimeConditions (Section 5.2) which verify whether the required preconditions are fulfilled at the beginning of the test execution. In a negative case, the test execution should be broken, since the tests cannot fulfill these conditions by themselves.

If automatic test executions are intended, much more data must be specified for correct test executions. Since the actual execution of generated tests is not in the focus of this work, we will not go into details about this data.

RuntimeCondition

As introduced in Section 2.2, an infotainment system HMI communicates with the underlying applications during runtime. The menu behavior during runtime is strongly dependent on the runtime data. For instance, a test first tries to find a particular mobile device via bluetooth, then connects it and makes a call from this device. The preconditions for the occurrence of the expected screens in this generated test are that the device can be successfully found and connected. However, during automatic test executions, there is no guarantee that a bluetooth device can be found every time. This leads to the problem that if the device is not found or not connectible for some reason, the test would report an error for the menu behavior, since the expected screen has not been displayed. In fact, this is not an error in the menu behavior but an error due to a missing runtime condition. This results in very time-consuming test result evaluations.

To solve this problem, the runtime data for the preconditions of an expectation must be verified during runtime. For this, type RuntimeCondition is defined.

Definition. *A* ***RuntimeCondition*** *is defined as a tuple $c = [n, f]$ where n is the name of the condition and the variables in function f should be bound to runtime data by the test instantiation.*

As introduced in Section 4.1, runtime data can be captured from interfaces such as the MOST-bus. In this way, a RuntimeCondition can be verified during runtime. If a RuntimeCondition is not fulfilled, a runtime error instead of a menu behavior error should be reported.

In the case that RuntimeCondition is required, the definition of menu behavior tests introduced in Section 4.2.2 must be extended. Steps = ($step_0$, ..., $step_k$) where $step_i$ with $0 \leq \mathrm{i} \leq \mathrm{k} \in$ AC $\cup$ EXP $\cup$ RuntimeCons where RuntimeCons is a set of RuntimeConditions.

Obviously, the differences between the defined types of conditions can be summarized as follows: PreStepsConditions can be fulfilled by the tests themselves during the test execution. PrepareConditions must be fulfilled before the execution of a test is started. RuntimeConditions are fulfilled during runtime by underlying applications.

Different event types

For automatic test executions, different event types must be defined to indicate to the test instantiation how to handle these events.

For instance, type ApplicationEvent is defined for events initialized by underlying applications such as the warning of an empty tank. Many menus require concrete user input data, e.g. a phone number to dial or a city name as in the example presented in Figure 5.8. Events such as ‘chooseCountry’ and ‘enterCity’ must be bound to concrete user data by the test instantiation. We have defined the type ‘UserInputEvent’ to indicate to the test instantiation that it should firstly check whether concrete user input data has been predefined and then bind the events with it.

Depending on the test automation framework used and the individual testing goals of different HMI testing projects, many more types of elements may be defined. In this thesis, automatic test execution is not our focal point. We merely present some challenges we have met and hope to give some inspiration to practical projects.

5.3. Requirements for the representation layer

The representation layer is designed to specify information about screens. Information about a screen contains firstly the contained UI elements and their properties, and secondly, events which can be triggered by the screen. In this section we will introduce how we reuse and combine existing approaches to specify this information about the screens.

5.3.1. Specification of screen contents

As introduced in Section 5.1, there are a number of specification languages for describing user interfaces. In this thesis, we neither develop a new specification language for user interfaces nor require a particular specification language. We only require certain data which must be specified for doing screen content tests. In order to introduce the requirements, we reuse the concept of *Model-based User Interface Development* (MBUID) [100] and some definitions from [100] which will be extended in this thesis. The focal point of MBUID is to develop the user interface.

The GUI development process in the MBUID contains the steps shown in Figure 5.9:

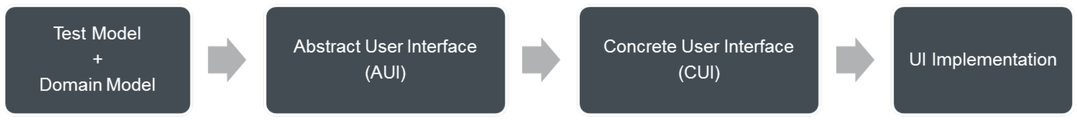


Figure 5.9.: The steps contained in a MBUID process

The *task model* and *domain model* are abstract models. The task model describes activities the user can perform with the UI in order to apply certain functions. The domain model describes the structure of the application logic.

"*The* ***abstract User Interface model (AUI)*** *specifies the user interface in terms of* ***Abstract Interaction Objects (AIO)*** *which are platform- and modality-independent abstractions of user interface elements.*" ([100]) Modality-independent means that different modalities such as graphics, speech and haptics are not distinguished for the AUIs and AIOs. There is still no standard notation for AUI.

The *Concrete User Interface model* (CUI) implements and refines the AUIs for a concrete modality and platform. As for the AUI, there is still no standard notation for CUI.

Finally, the *UI implementation* is generated from the CUI model.

In order to abstractly specify the screen contents for performing screen content tests, we reuse the concept of the abstract user interface model: AUI. In the AUI, a **presentation unit** is an abstract form of a screen, into which AIOs are grouped. As yet, there is no common standard notation for AUI models. The following class diagram presents the relationship between a representation layer and its contained AIOs.

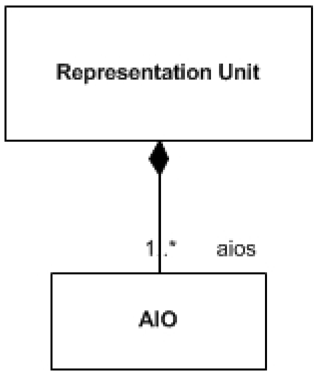


Figure 5.10.: Representation unit and contained AIOs

We concretize and extend this concept for our representation layer as follows:

Definition. *A* ***representation unit*** *in the test-oriented HMI specification is an abstract form of a concrete infotainment system screen. A representation unit contains at least one AIO.*

Definition. *An Abstract Interaction Object (**AIO**) in the test-oriented HMI specification is an abstract form of an UI element which has properties which can be tested via screen content tests. The properties can be specified as attributes of the AIOs.*

For the screen 'NavAdrInput' presented in Figure 2.4, the following representation unit can be specified with the contained 'country', 'city', 'street' and 'start' elements:

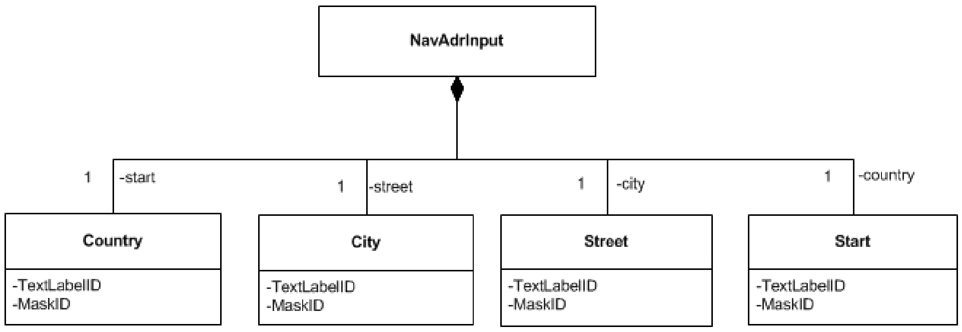


Figure 5.11.: Representation unit for the screen NavAdrInput

The property "TextLabelID" is a reference to a specified text in the text layer as shown in Figure 5.2. The property "MaskID" is a reference to a defined mask in the design layer as presented in 5.3. For each of the specified UI elements, the screen content test should verify whether the given text occurs in the given mask.

The UI elements specified above have static positions. However in practice, there are also some UI elements with dynamic positions which are dependent on the runtime data. For instance, depending on the number of given stopovers for the navigation guidance, the widget 'start' moves accordingly downward. The position specification of such widgets must be bound to runtime data, as for the menu behavior introduced in Section 5.2. However, this is not a focal point of this thesis and will not be discussed further.

5.3.2. Specification of possible triggerable events

A screen can trigger events in reaction to user inputs. For instance, if the user presses the ‘country’ button in the ‘NavAdrInput’ screen shown in Figure 2.5, the screen fires the ‘country’ event, which triggers the transition in the menu behavior model as shown in Figure 5.8. The definition of the events which can be triggered by a screen is required to calculate the semantics of screen-dependent transitions.

There are different ways to specify possible events which can be triggered by screens. One of them is described in the work [16], which is based on an abstract screen model. An abstract screen model contains abstract screens for which all possible triggerable events are specified. Figure 5.12 presents the relationship between abstract screens and triggerable events.

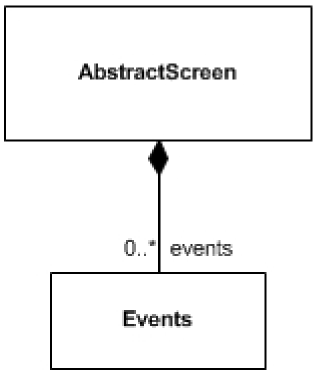


Figure 5.12.: Abstract screen and events which can be triggered

In [16], potential events are specified as “pins” of an abstract screen. Figure 5.13 presents an example of the abstract screen ‘ConfirmScreen’ and the event ‘yes’ which can be triggered from this screen.

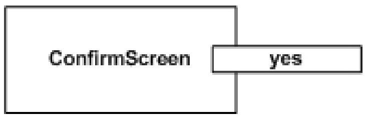


Figure 5.13.: An abstract screen defined in [16]

In [16], an abstract screen can inherit another abstract screen, as shown in Figure 5.14. The screen ‘ConfirmScreenNo’ can trigger both the event ‘yes’, which is inherited, and the event ‘no’. This inheritance concept is used in order to simplify the specification of events.

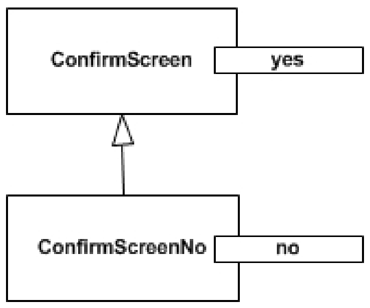


Figure 5.14.: Inheritance in the abstract screen model

This concept will be reused in this thesis in order to specify the potential events which can be triggered by screens.

5.3.3. Meta-model of the representation layer

In Section 5.3.1 and Section 5.3.2, existing approaches which are to be reused in this thesis have been introduced. In this section, we will introduce the meta-model for the representation layer. The meta-model defines the data which is required for testing and should be specified for screens in the representation layer.

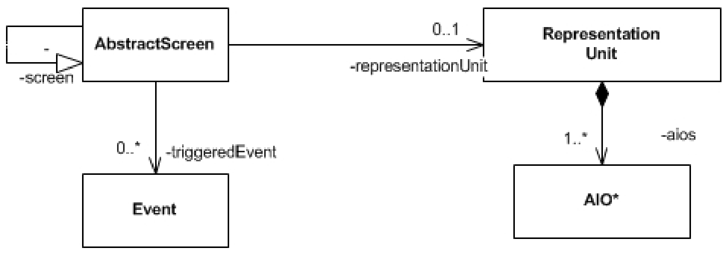


Figure 5.15.: Meta-model for the representation layer

As shown in Figure 5.15, the representation layer of the test-oriented HMI specification contains abstract screens. For each abstract screen, events which can be triggered are specified. An abstract screen is also associated with a representation unit, which describes the screen contents, thus the contained UI elements and their properties which are to be tested.

Figure 5.16 shows the specification of screen ‘NavAdrInput’:

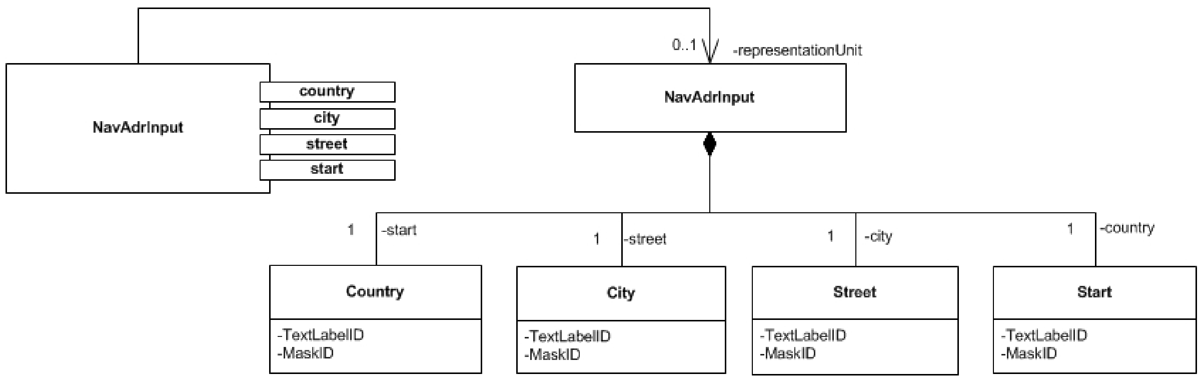


Figure 5.16.: Specification of screen NavAdrInput

As introduced in Subsection 5.2, a view state is associated with an abstract screen defined in the representation layer. For instance, the view state 'NavAdrInput' in Figure 5.8 is associated with the abstract screen shown in Figure 5.16.

5.4. Related work

The test-oriented HMI specification describes the HMI in terms of the dynamic menu behavior and the representation. In this section, related work in the following domains will be introduced:

1. Development and testing architectures/frameworks in which GUIs can be modeled
2. GUI-development tools in which GUIs can be modeled
3. Specification languages with which GUIs can be modeled

A meta-modeling architecture was developed which supports the design and development of interactive software systems within the European project **CAMELEON** [URLb] (Context Aware Modeling for Enabling and Leveraging Effective interaction). In this architecture, different types of models are used to describe different relevant contents. For the automatic generation of GUI software, the GUI is described with the following abstract models which are separated from each other: task and concept models, the abstract user interface model, the concrete user interface model and the final user interface model [71]. Like the AUI, which was introduced earlier in this chapter, the abstract interface model in the CAMELEON framework describes the GUI abstractly and independently of a modality and platform. It can be obtained from the tasks and concept models via transformations. Gerrit Meixner has derived an architecture for the model-based useware-development process from the CAMELEON framework in his doctoral thesis [77]. In this, the abstract user interface model is composed of an abstract presentation model and an abstract dialog model. In [85], he proposed the "dialog and interface specification language" (DISL) for the specification of the representation and the dialog behavior. The framework **MARIA** [98] supports the approach of CAMELEON and allows the user to describe the UI on both an abstract and a concrete level.The language used for the abstract UI description is independent of a platform and modality. MARIA provides a number of languages which can be used to concretize the abstract description for particular platforms and modalities. Many other frameworks in which GUIs can be modeled have already been introduced in Chapter 3.

Several industrial tools are developed for the model-based GUI development: **EB GUIDE Studio** [URLf], **IRIS** [URLg], **VAPS XT** [URLi] and **Altia&Rhapsody** [URLa]. All of these tools allow the creation of concrete screens and the modeling of the behavior. However, these tools do not support abstract description of the screens, since they are GUI development and prototyping tools and do not focus on an abstract specification. The tool **TERESA** [14] supports the approach of CAMELEON and allows the design and development of multi-platform applications. An abstract specification of the UI is supported in this tool. However, TERESA is not supported anymore. Instead, the **MARIAE** (MARIA Environment) tool is proposed. This is based on the same concept as TERESA, uses the MARIA language and provides additional functions such as the automatic import of services and annotations.

A number of specification languages support the abstract or concrete description of GUIs.

The Original Equipment Manufacturer XML (**OEM XML**) [7] is especially developed for the automotive HMI domain. An abstract description of the HMI is supported. The complete structure of the HMI is described as a tree in which each node corresponds to a widget, while the structure and the behavior of a single widget are described in one or more separate XML files. In this specification language, the description of the behavior is suboptimal. Similarly to the modeling concept introduced in [108], the description of HMI states are dependent on the widgets; in order to model a state, a corresponding widget must first be described.

Elektrobit [URLe] has developed an HMI specification language **ICUC XML** for Daimler AG which supports HMI development for trucks. The main purpose of this language is to allow the storage and transmission of the contents from the HMI development models (Section 2.2.3) which are created with EB GUIDE Studio. Therefore, the language is not anticipated as being useful for abstract descriptions. The descriptions of the screens and the behavior are separate from one another.

The Extensible User Interface Language (**XUL**) [17] [32] is developed by the Mozilla project for developing cross-platform applications. The goal is to allow the porting of applications for different platforms on which Mozilla applications run. XUL can be used for describing window layouts. The GUI is described with a structure which contains the graphical elements such as windows and buttons. However, XUL is not complied with specifying dynamic behaviors.

The User Interface Markup Language (**UIML**) [99] was developed to describe the GUI independently of any particular programming language. It allows the separation of the representation and the behavior descriptions. Interpreters are used to translate the GUI elements in UIML into components of the used language such as Java or HTML. The drawback of this language is that it describes the behavior based on a rule-based system where events are included in the conditions. In comparison to state-based descriptions as in many other XML-based languages, it is quite unusual to describe a state-oriented system such as the HMI with a rule-based system.

There are a number of other XML-based specification languages, e.g. the State Chart XML (**SCXML**) [URLj] which can only describe the behavior, the Extensible Application Markup Language (**XAML**) [73] which is used to simplify the creation of GUIs for .NET applications,

Adobe Flex 3 which is suitable for rapid prototyping, and the Presentation Markup Language (**PreML**) which is only conditionally suitable for modeling the representation etc. There are also efforts aimed at new specification languages which combine the advantages of these existing languages and optimize them for modeling the infotainment system HMIs in an abstract way, such as [118] and the automotiveHMI project [URLd]. Each of these modeling languages has its advantages and drawbacks. A detailed comparison of many existing XML-based specification languages can be found in [110] and [41]. We will not introduce all of them in detail here since specification languages are not within the focus of this thesis.

5.5. Conclusions

In the last chapter it was stated that the main focal point of this thesis is testing the menu behavior. Screen content tests should also be addressed. The precondition to perform these tests is that the menu behavior and the screen contents which are to be tested are specified in the test-oriented HMI specification. In this chapter, the test-oriented HMI specification was introduced, in particular the menu behavior model and the representation layer, which are the main components of the test-oriented HMI specification. Requirements have been defined for data which must be specified in the menu behavior model and the representation layer. The test-oriented HMI specification is now ready for test generation.

Chapter 6.

Test Generation

As explained in Section 2.1, test generation is the process in which tests are automatically generated from one or several test models based on particular coverage criteria. In our context, menu behavior tests should be generated from the menu behavior SC which is the test model. Currently, there already exist a number of test generation techniques (Section 6.4). In this thesis, we do not aim to develop a new test generation technique, but only to demonstrate how menu behavior tests can be generated from the menu behavior SC based on three conventional structural coverage criteria: transition coverage, path coverage and state coverage (Section 2.1.2), which are adequate for state-based models. Generated tests based on these coverage criteria in particular will be presented and compared. The advantages and drawbacks of using these coverage criteria will be discussed. First of all, we introduce the example menu behavior SC from which tests should be generated.

6.1. The menu behavior SC

Figure 6.1 shows the menu behavior SC from which tests should be generated. This SC describes a subset of the menu behavior of the navigation function in a flat structure. In practice, a menu behavior SC usually has a large set of hierarchies due to the high complexity. However, we propose to allow the generation algorithm to be based on a SC model with a flat structure and flatten the hierarchical SC models before tests are generated from them. This firstly keeps the generation algorithm simple, adaptive and less error-prone, and secondly makes the test generation more efficient, since it would be time-consuming if the test generation algorithm had to resolve the same hierarchies for each generation activity. The process in which the hierarchies are resolved is called model flattening [107]. It must ensure that the resultant flattened SC model maintains the semantics of the original hierarchical SC.

The SC above describes the following menu behavior: the system starts with the screen 'Main'. From 'Main', the user has the option to launch the navigation function, if the PrepareCondition "Navi is installed" is fulfilled, as introduced in Section 5.2. On choosing the navigation function, the user is led to the 'NavWizard' screen which offers the options to either define a destination by entering an address or access the address book which contains predefined contact addresses. The second option is not completely described in this SC. For the first option, the user must first choose a country from the country list screen. After that, he can input either a city followed by a street or a street only, and then start the route guidance. This SC is extremely simplified in

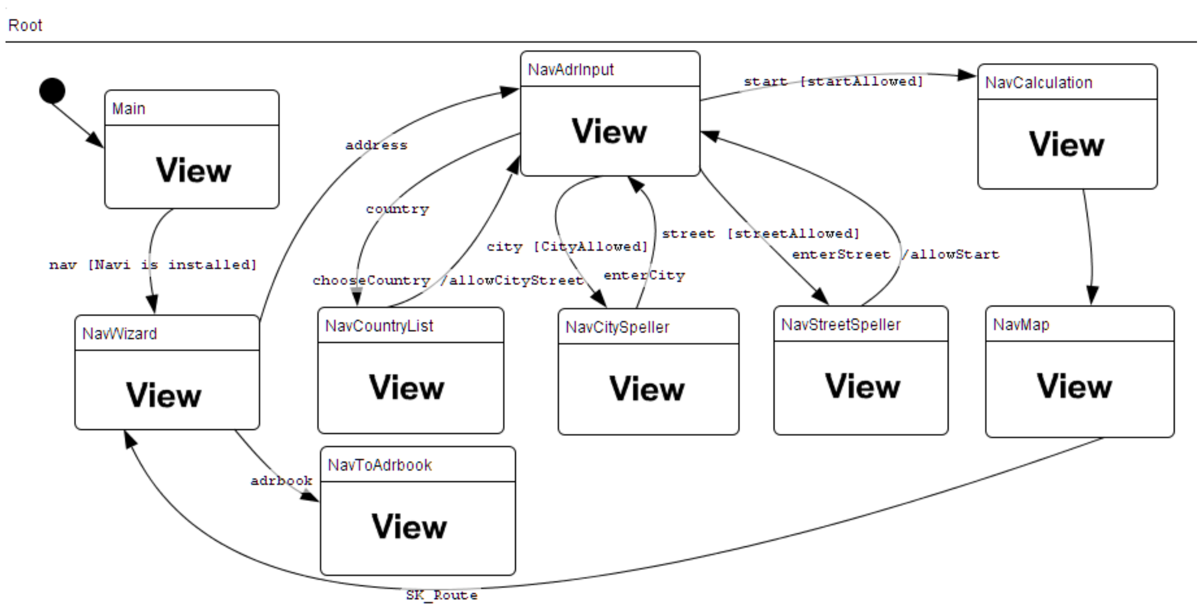


Figure 6.1.: Flattened SC model for test generation

order to keep the introduction easy. In practice, the menu behavior of the navigation function is much more complicated and provides many more options for defining a navigation goal.

6.2. A test generation algorithm

In this section, we introduce in detail a test generation algorithm which is based on the transition coverage. This algorithm traverses the SC model based on the depth-first search and systematically generates the possible paths through the SC. It allows each cycle only once in order to avoid test case explosion [70]. The principle of the test generation algorithm can be described with the following rules:

- The generation algorithm begins with the initial state
- The generation algorithm carries out a depth-first search and systematically generates the possible paths through the SC. Each path is a test
- A cycle is only allowed once. This rule exists in order to avoid test case explosion
- If the test generation visits a state which has no outgoing transition, it closes the current path and continues with other unclosed paths if any are available
- If the test generation visits a state whose outgoing transitions have all been visited already, it closes the current path and continues with other unclosed paths if any are available

- If the test generation visits a state which has only one unvisited outgoing transition from which the condition is fulfilled, it adds this transition and the next state to the end of the current path and continues with this current path
- If the test generation visits a state which has more than one unvisited transition from which the conditions are fulfilled, it continues with the first unvisited transition and marks the other unvisited transitions in order to visit them later
- After the test generation has inserted a transition and its target state into a path, it verifies whether the chosen test coverage (in this case the transition coverage) is fulfilled. If all transitions have already been visited, it stops the test generation

The following pseudo-code describes one possible implementation of the algorithm:

```
generateTests(statechart sc, coverage criteria cc)
begin
    initialize a test curTest, a list of tests testList and a state startingState;
    set curTest.status = inProcessing;
    testList.add(curTest);
    set startingState = initial(sc);
    generateRec(curTest, testList, startingState, cc);
end
```

```
generateRec(test curTest, list testList, state startingState, coverage criteria cc)
begin
    //For the first invocation the starting state is the initial state
    //For later invocations overwrite the startingState
    if curTest $\neq$ null then startingState = curTest.getLastState();

    //Case 1. If current starting state has no outgoing transition
    if outgoingTransitions(startingState) = $\emptyset$ then
        set curTest.status = finished;
        if testList contains at least one test with status(test)=inProcessing then
            //Get the first test with status "inProcessing" from the list
            curTest = testList.getFirstTest(inProcessing);
            generateRec(curTest, testList, startingState, cc);
        else return testList;

    //Case 2. If all outgoing transitions of the current states are already visited
    if for each tr $\in$ outgoingTransitions(startingState): status(tr)=visited then
        set curTest.status = finished;
        if testList contains at least one test with status(test)=inProcessing then
            curTest=testList.getFirstTest(inProcessing);
            generateRec(curTest, testList, startingState, cc);
        else return testList;

    //Case 3. If current starting state has one unvisited outgoing transition
    let n = the number of transitions in outgoingTransitions(startingState) with
    status(tr) = unvisited and cond(tr) = true;
    if n = 1 then
        curTest.insert(tr);
        curTest.insert(target(tr));
        if cc is fulfilled then return testList;
        else generateRec(curTest, testList, startingState, cc);

    //Case 4. If current starting state has more than 1 unv.. outg. transitions
    let be T the set of transitions in outgoingTransitions(startingState) with
    status(tr) = unvisited and cond(tr) = true; let n = T.size();
    if n > 1 then
        copy curTest for n-1 times and let TST be the set of copied tests;
        for the k-th transition tr $\in$ T and k-th test tst $\in$ TST with $0 \leq k \leq$ TST.size()-1:
            tst.insert(tr); tst.insert(target(tr));
            testList.add(tst) and set tst.status = inProcessing;
        for the last tr $\in$ T: curTest.insert(tr); curTest.insert(target(tr));
        if cc is fulfilled then return testList;
        else generateRec(curTest, testList, startingState, cc);
end
```

This implementation uses the above algorithm. A specialty of this implementation is that it implements the mark function which is required if a state is visited which has more than one unvisited transitions (Case 4), based on the copy path principle, i.e. if the test generation is in the view state ‘NavWizard’, only one path has been generated so far: Path 1 = [Main] → (nav) → [NavWizard]. The test generation detects that the view state NavWizard has two unvisited outgoing transitions, then creates a new path Path 2 by copying Path 1, adds one of the unvisited outgoing transitions (e.g. the one which is labeled with event ‘address’) to Path 1, adds the other transition to Path 2, marks Path 2 as “inProcessing” and continues with Path 1. Later, when Path 1 is finished, all paths with the status “inProcessing” are processed.

This is a quite an unusual implementation. But it has the advantage that it avoids “outlier tests”: tests which are much longer than the others. We have also implemented the mark function in the classical way by marking the actual unvisited transitions as “still unvisited” instead of copying the complete path. This implementation generates different tests from the first implementation. It generates one “outlier test”. This will be shown in the next section. In [105], the point is also raised that different implementations of the same test generation algorithm based on the same coverage criteria usually produce very different tests. This is not contrary to the determination of a test generator. Each test generator is deterministic and produces the same results for the same model. However, different generators which implement the same generation algorithm and are based on the same coverage criteria usually implement the multitude of details in different ways, usually leading to different results.

In order to demonstrate the correctness of the implementation, we have generated tests from several menu behavior SCs and verified whether the generated tests are valid and cover all transitions.

6.3. Generated tests

6.3.1. Based on the transition coverage

The implementation which was described with pseudo-code generates the following three tests which have the same condition set Cons = {[Navi is installed]}:

Test 1:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput]

Test 2:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap] → (SK_Route) → [NavWizard]

Test 3:
[Main] → (nav) → [NavWizard] → (adrbook) → (NavToAdrbook)

The tests produced have comparable lengths due to the copy path principle. However, this test generation leads to the result that the typical way of entering a destination, i.e. first country, then city and finally street, is not covered. However, the generated tests fulfill the transition coverage.

The second implementation introduced above produces the following two tests:

Test 1:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap] → (SK_Route) → [NavWizard]

Test 2:
[Main] → (nav) → [NavWizard] → (adrbook) → (NavToAdrbook)

These two tests also cover all transitions in the SC. The classical way of defining a destination is covered by Test 1. The length of Test 1 is unproblematic. However, from a larger menu behavior SC describing a very complex behavior, very long tests would be generated with the drawback that if they are broken somewhere at the beginning, e.g. due to an unfulfilled condition, the remaining steps can no longer be verified.

6.3.2. Based on the path coverage

In order to verify the possible scenarios of entering a destination and starting the route guidance, we have implemented an algorithm which is based on the path coverage. As introduced in Section 2.1.2, path coverage could lead to non-terminating generation if infinite paths exist in the model. Therefore, we limit the number of generated tests using the following two strategies: first, each cycle is allowed only once (as we have done so far), and second, the algorithm should only generate tests which end with the view state 'NavMap'. This view state is chosen since our goal is to test the paths which enable the initiation of the guidance. The view state 'NavMap' is expected as soon as the guidance is started. The following tests are generated:

Test 1:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap]

Test 2:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap]

Test 3:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap]

Obviously, all three possible paths to starting the route guidance are generated. Test generation from a more complex menu behavior SC has shown that most of the generated tests share a large set of common step sequences. The same errors are repeatedly detected by many of these tests. This means that the execution of more tests does not necessarily mean the detection of more errors, but merely consumes time resources. Therefore, the path coverage is only suitable during less demanding project phases.

6.3.3. Based on the state coverage

The test generators which are based on the two different implementations of the mark function separately produce the following tests based on the state coverage :

Test 1:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput]

Test 2:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap]

Test 3:
[Main] → (nav) → [NavWizard] → (adrbook) → (NavToAdrbook)

and

Test 1:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap]

Test 2:
[Main] → (nav) → [NavWizard] → (adrbook) → (NavToAdrbook)

These tests together cover all states in the SC. In comparison to the results which were produced based on the transition coverage, these tests are identical except that the transition with the event ‘SK_Route’ is not covered and the view state ‘NavWizard’ is not visited again via this transition. This is because the state coverage is not as strong as the transition coverage; in cases in which a state has more than one incoming transition, the state is usually covered via one of them while the other transitions remain unvisited.

For a menu behavior SC, state coverage implies that each view state must be covered by the generated tests. This is especially helpful for testing foreign language systems. Using screenshots captured from each screen visited during the test execution, the screens can be easily reviewed by native speakers for language verification without the need for test benches/cars or manual executions of a generally very large number of tests.

6.4. Related work

In this chapter, a test generation algorithm was introduced which generates tests from a menu behavior SC based on the depth-first search and transition coverage. A large number of test generation methods can be found in related research literature. These methods differ from each other in the applied generation techniques, model types (state-based or event-based), specification languages used, and chosen coverage criteria. In this section we introduce some of them.

In [51], the authors have proposed the **transformation** of the **UML SC** model into a flow graph which is based on Extended Finite State Machines (**EFSM**). The flow graph describes both the control flow and the data flow which are described in the SC. The transformation has the advantage that conventional control and data flow analysis techniques and adequate coverage criteria can be applied for the test selection.

A large number of efforts address the test generation from **FSM**. A survey of the problems, principles and methods of testing FSM can be found in [67].

In [38], elements in a **UML SC** are mapped into STRIPS, a **planning language**. In this way, the test generation problem is transformed into the planning problem. With the state-of-the-art planning tool *graphplan*, test cases (which are sequences of messages) and test data can automatically generated. This planning technique ensures that only those test sequences are generated whose preconditions are satisfied. A generation algorithm which is also based on the transition coverage has been demonstrated.

In [52], **model checking** is applied on **UML SCs** in order to perform test generation. The problem of test generation is transformed into the problem of finding counter-examples during model checking. Control flow oriented coverage criteria, i.e. state coverage, configuration coverage and transition coverage, and data flow oriented coverage criteria, i.e. all-def coverage and all-use coverage, can be applied to the test generation which is based on the model checking technique.

In [58], a method was introduced for generating test cases from an Input/Output-Pairs Label Transitions System (**IOLTS**) based on model checking. In [43] too, the **UML SC** is transformed into an IOLTS in order to apply existing LTS test generation algorithms.

Depending on the model types, such as state-based or event-based, and the specification languages used, different **coverage criteria** have been defined. For instance, in [91], general criteria for generating tests from state-based specifications have been proposed. In [83], coverage criteria are introduced which are adequate for generating tests from event-flow graphs.

Basically, for the generation of HMI tests from our menu behavior SC, any generating technique can be applied which is suitable SC models. The main point is that the test generation must be extended to consider the elements which are used in the menu behavior SC, e.g. view states and PreStepsConditions, and ensure the generation of valid tests. Any coverage criteria which are adequate for state-based models can be applied.

6.5. Conclusions

In this section, it was introduced how menu behavior tests can be generated from a menu behavior SC based on three conventional structural coverage criteria. An algorithm was introduced which is based on the depth-first search and the transitions coverage. In contrast to many other test generation methods, the algorithm traverses the menu behavior SC and directly generates tests from the SC by selecting paths without model transformation, model checking, or any other complex techniques. One of the implementations of this algorithm was demonstrated as pseudo-code. Also, some advantages and disadvantages of the three different coverage criteria were discussed.

So far, we have not yet focused on the variability of infotainment system HMIs. The introduced algorithm and implementations serve as the basis for the extensions for testing under variability which will be introduced in the next section. It will be shown how this algorithm can be extended to take SC elements representing variability into account and generate tests for different variants at one stroke.

Part III.

Integrating Variability Into Model-Based HMI Testing

As explained in the introduction, infotainment system HMIs have a special characteristic: variability. Variability can be a result of product series such as different generations, market variants such as the European and Chinese markets, configuration variants such as with or without DVD player, and system variants such as standard display or display. In practice, there can be up to 100 HMI variants in the same development project. Variability poses a huge challenge to the testing process.

Different variants of the HMI share a large set of commonalities in functions, looks and behaviors. Due to these commonalities, it makes sense to regard these variants as a product family and hence specify and test them as products of a product family based on software product-line approaches [101] [66]. In the software product-line domain, the term "variant" (which we have used until now) is usually replaced with the term "product". Therefore, from now on, the term "HMI product" is used as a synonym for "HMI variant".

In this section, the proposed test-oriented HMI specification and generation method in Part II will be extended for testing under variability. First, we introduce how variability can be integrated into the test-oriented HMI specification. Afterwards, it will be shown how menu behavior tests can be generated for different HMI products to be tested from such a menu behavior model. The particular aim of this generation method is to avoid redundant generation activities. Finally, a solution will be provided for avoiding identical tests that verify the common behavior of different HMI products.

Chapter 7.

HMI Specification With Variability

In this chapter, we describe how variability can be integrated into a test-oriented HMI specification. To do this, we aim to answer the following questions:

- how to manage variability
- how to extend the menu behavior model to describe variability
- how to extend the representation to describe variability

A software product-line, also known as a product family, or in short SPL or PL, is a very widely used approach to developing products that share commonalities. In [23], the software product-line is defined as follows:

A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

In order to test the variability of HMIs, our approach is also based on the concept of the product-line. This means we assume that the different HMI variants to be tested share a large set of commonalities. We regard the composition of these variants as a product family and hence specify and test them as products of a product line. Both the commonalities and the individuality of these HMI variants are specified in one test-oriented HMI specification.

Since the term "product" is usually used in this domain instead of "variant", we also prefer the term "HMI product" to "HMI variant", which has been used until now. In literature, there are a number of existing SPL development and testing approaches. However, due to the special characteristics and development processes of infotainment system HMIs, different problems must be faced in our work. In Section 7.4, we will compare the challenges and solutions of our work and some related work.

Figure 7.1 shows three HMI products provided by a product line. This product line will be used as the ongoing example for this chapter. This example is highly simplified from a realistic product line.

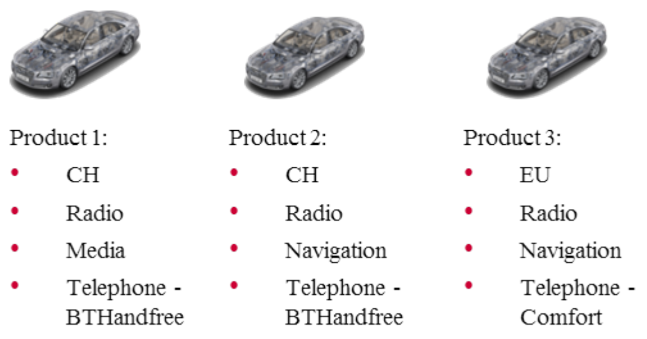


Figure 7.1.: Three HMI products of a product line

The product line provides four functions: radio, media, navigation and telephone. The radio function is the basic function provided by all products of the product line. Other functions are configurable functions which can be individually chosen in the product. The telephone function can be provided with either the “BTHandsfree” (Bluetooth Hands Free) or “comfort” option. The “BTHandsfree” option provides the possibility to connect the mobile phone with the car via a bluetooth interface and to make a call using the SIM card of the mobile phone via the amplifiers and the microphone of the car. “Comfort” provides an integrated telephone in the car which can be connected with the SIM card of the mobile phone (also via a bluetooth interface). In this case, the mobile phone is in standby mode. A call is directly made by the in-car telephone. The product line also provided products for two markets: the European market (“EU”) and the Chinese market (“CH”).

7.1. Feature specification

7.1.1. Extended feature model

The first problem we have to address in testing the HMI variability is how to describe the product line and manage variability in a test-oriented HMI specification. To do this, we extend the feature model that is very widely used in SPL for describing the product line and its products.

Feature models, or FMs for short ([44] [6] [25]) allow one to describe both the commonalities and variabilities of products of a PL and to define relationships between the provided features. A FM can be described graphically as a tree, in which the nodes represent the features and the edges represent the relations between them. Four types of relations exist between features: mandatory, optional, alternative and disjunct. Constraints can be defined to describe the relationships between the features, e.g. implies-, and-, or- and not- relations.

Several notations are available for describing FMs, such as the one described in [59] and [25]. In this work we will use the notation which was proposed by Czarnecki in [25].

Definition. *A* ***feature model*** *is defined as a tuple (Funcs, f_0, Mand, Opt, Alt, Or-rel) where Funcs is a set of functionalities of a domain (nodes of the tree), $f_0 \in$ Funcs is the root* *functionality of the tree and $Mand$, Opt, Alt, Or-rel the mandatory, optional, alternative and disjunct relations of the model.*

A feature model configuration (FMConf) is an instance of a FM. It respects the relations and constraints in the FM and describes a concrete product.

Definition. *A **feature model configuration** (FMConf) corresponding to a FM (Funcs, f_0, $Mand$, Opt, Alt, Or-rel) is defined as a tuple FMConf = (F, R) which can be presented as a tree. $F \subseteq Funcs$ is the set of nodes and R is the set of edges. FMConf respects the relations and constraints defined in the FM.*

A classical feature model contains only one type of feature in terms of functions which are provided by the described product line. Different products of a product line differ from each other only in the provided functions. However, HMI products of a product line are characterized by features not only in terms of functions such as navigation and telephone, but also in terms of variants such as market orientation (e.g. European or Chinese). Also, a function can be provided with different options such as the telephone options "BTHandfree" or "comfort". These different feature types must be specified in different layers and in different ways in the test-oriented HMI specification. The test generation too must treat them differently. Therefore, classical feature models must be extended for describing an HMI product line; the differentiation between different types of features must be included.

We extend the introduced feature model into the so-called extended feature model as follows:

Definition. *A **extended feature model** (EFM) is defined as a tuple ($FFeatures$, $VFeatures$, f_{root}, $Nodes$, Mand, Opt, Alt, Or-rel) which can be presented as a tree. $FFeatures$ is a set of functional features. A functional feature $ff \in FFeatures$ can be atomic and is presented as a leaf in the tree or it has a set of functional feature options FFOptions(ff) which are presented as the leaves in the tree (ff as their parent node). $VFeatures$ is a set of variant features which can be presented as leaves in the tree. $FFeatures \cap VFeatures = \emptyset$; f_{root} is the root of the tree; $Nodes$ is the set of the remaining nodes in the tree;*

A functional feature corresponds to a function provided by the infotainment system. A functional feature option corresponds to the options with which a functional feature can be provided. A variant feature represents a characteristic of a product which is non-functional.

Also for an EFM, constraints can be defined to describe the relationships between functional features, functional feature options and variant features. Logic operators $\wedge$ (and), $\vee$ (or), $\Rightarrow$(implies) and $\neg$ (not) can be used to define the constraints.

Figure 7.2 shows the EFM describing the simplified product line displayed in Figure 7.1. The radio, media, navigation and telephone features are functional features. Of these, the radio is contained in each product of the PL and the others are optional. Functional features are labeled with "f" in the EFM tree. The functional feature state telephone has two options: "BTHandfree" or "comfort". These functional feature options are labeled with "o" in the tree. The products of this product line can be either for the market variant Europe (EU) or China (CH), which are variant features. Variant features are labeled with "v" in the tree.

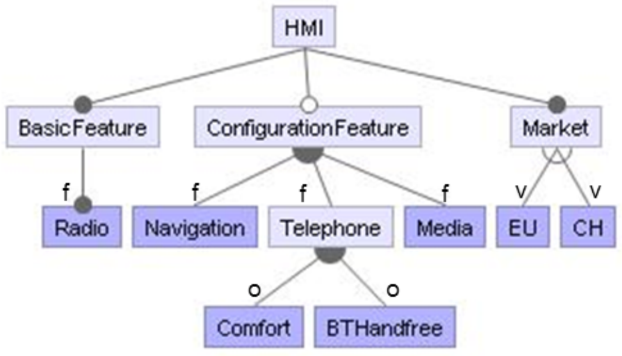


Figure 7.2.: Extended feature model describing a simplified HMI product line

Constraints can be defined between functional features, variant features and optional features. For instance, the rules"CH implies BTHandfree" and "EU implies navigation" constrain the products which can be provided by the product line in Figure 7.2.

An EFMConfig is an instance of the corresponding EFM and describes an HMI product of the PL.

Definition. *An **extended feature model configuration** (EFMConf) corresponding to an EFM = ($FFeatures$, $VFeatures$, f_{root}, $Nodes$, Mand, Opt, Alt, Or-rel) is defined as a tuple (FF, VF, N, R) which can be presented as a tree. $FF \subseteq FFeatures$ is the set of functional features provided in the described product, $VF \subseteq VFeatures$ is the set of variant features of the described product. $N \subseteq Nodes$ is the set of remaining nodes and R is the set of edges. EFMConf respects the releations and constraints defined in the EFM.*

7.1.2. Related work

The idea of using feature models for describing the commonalities and variabilities among products of a product line is not new. Basic principles and notions describing FMs can be found in [59] and [25]. The authors of [68] have had wide experience of feature modeling used in several industrial product line projects. They have provided very solid analysis about what a feature is, as well as an overview of feature modeling and guidelines for feature modeling.

In [44] a method was introduced for how feature models can be used together with UML SC to describe variability in a behavior model. In this work, the original form of feature models is used. An example has been shown which describes a very simple product line. Various attempts have been made to extend feature models and apply them for the individual domains. In [34], the feature model is extended for the context-aware software product line. For this, context information and adaptation knowledge are incorporated into the feature model via modeling notions. Also, a mechanism which "*verifies the consistency of a product configuration regarding context variations*" is extended into the feature model. In [8], the structure and the constraints of a feature model are extended in order to "*allow easy integration and satisfaction of the stakeholder's soft and hard constraints, and the application-domain integrity constraints*".

7.2. Menu behavior model with variability

Different HMI products to be tested based on the same product line usually share a large set of common menu behaviors. They also have individualities in the menu behavior. In order to test the menu behavior of these products based on product-line approaches, both the commonalities and the individualities must be specified in the menu behavior model of a test-oriented HMI specification.

In this section, we introduce how the specification of variability can be extended into the menu behavior model which is a SC in this work. In our approach, both the commonalities and the individualities should be described in the same menu behavior SC. In literature, there are also approaches in which the commonalities are described in one SC and each commonality is described in a separate SC. This type of approach will be introduced in Section 7.4.

Different products to be tested in a PL differ to each other in the functional features, functional feature options and the variant features. Therefore, in order to specify variability in the menu behavior SC, the parts of the menu behavior describing different functional features, functional feature options and variant features must be distinguishable from each other.

A menu behavior SC describing variability is defined as SC* (SC with variability) in this thesis. The following is the definition of SC*:

Definition. *A* ***SC**** *is defined as a tuple SC* = (S, $FStates$, $VPoints$, $JPoints$, TR, E and A ($E \subseteq A$)) where $FStates$ is the set of functional feature states, $VPoints$ is the set of variation points and $JPoints$ the set of joining points; $FStates \cup VPoints \cup JPoints \subset S$; the remaining elements have the same definition as in SC.*

Functional feature states, variation and joining points will be introduced in the next sections.

7.2.1. How to incorporate functional features into the menu behavior SC

In order to incorporate the functional features into the menu behavior SC, the following elements must be defined.

Functional feature state

An infotainment system is a service-oriented system. The services provided can usually be clearly divided into functions such as media, navigation, etc. The menu behavior of the HMI can describe the menu behavior of these functions. Therefore, the complete menu behavior can usually also be clearly divided into functional features (Figure 7.2), which correspond to the provided functions of the infotainment system.

In order to separate the behavior describing different functional features in one SC, the type functional feature state is defined:

Definition. *A **functional feature state** is an or-state defined as $ff\_state = [n, (s_0, ..., s_k), T, ff]$, where ff ∈ FFeatures is the corresponding functional feature defined in the EFM, substates $s_0, ..., s_k$ and all transitions in T describe together the menu behavior of ff.*

Each functional feature state $ff\_state \in FStates$ describes the menu behavior of a unique functional feature ff ∈ FFeatures defined in the EFM. And the other way around, for each functional feature ff ∈ FFeatures in the EFM, there must be a functional feature state $ff\_state \in FStates$ in the corresponding SC* describing its menu behavior.

Function. *For a functional feature state ff_state in FStates, **FunctionalFeature(ff_state)** = ff ∈ FFeatures obtains the associated functional feature defined in the EFM.*

Function. *For a functional feature ff ∈ FFeatures, **FunctionalFeatureState(ff)** = ff_state ∈ FStates obtains the functional feature state in the SC*, which describes the behavior of ff*

As shown in Figure 7.3, 'RadioRoot', 'MediaRoot', 'NavigationRoot' and 'TelephoneRoot' are functional feature states that respectively describe the menu behavior of the functional features radio, media, navigation and telephone defined in the EFM in Figure 7.2. Usually, functional feature states are directly located under the root state in the SC*.

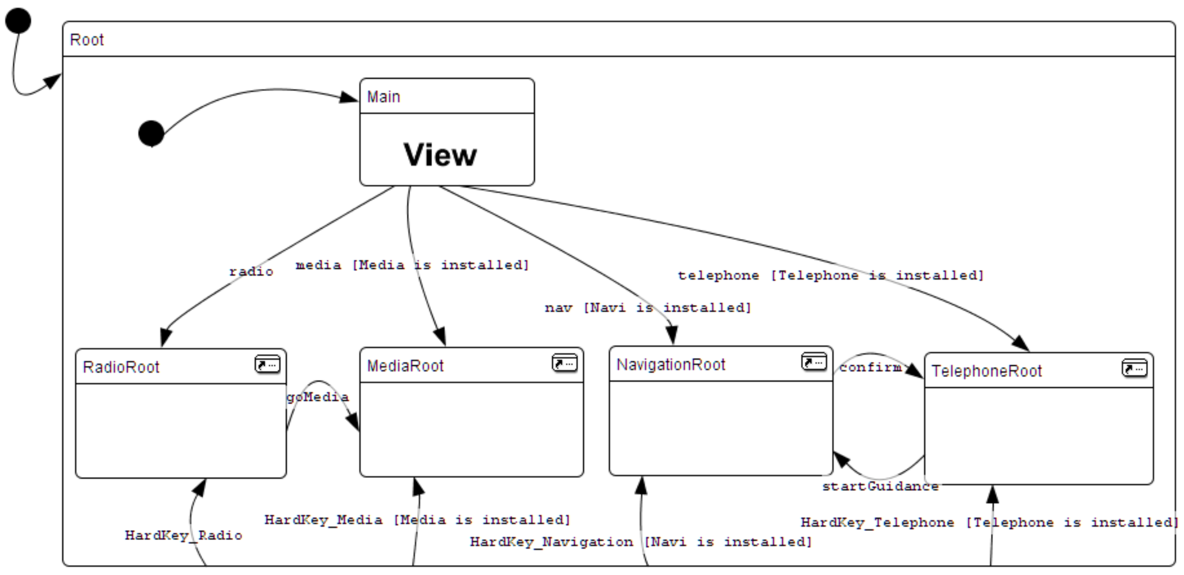

Root
Main
View
radio
media [Media is installed]
telephone [Telephone is installed]
nav [Navi is installed]
RadioRoot
MediaRoot
NavigationRoot
confirm
TelephoneRoot
goMedia
startGuidance
HardKey_Radio
HardKey_Media [Media is installed]
HardKey_Navigation [Navi is installed]
HardKey_Telephone [Telephone is installed]

Figure 7.3.: Functional feature state of SC*

Sometimes, it is not explicit whether a service provided by the infotainment system should be regarded as merely a "sub-function" of another function or as an autonomous function which has relations and connections to the other function, e.g. the navigation function settings versus the navigation function. 'Connection' here means that the user can access the navigation function settings from the navigation function and the other way around. Basically, both these forms are feasible in our concept; if a navigation setting is regarded as an autonomous function, then a functional feature must be defined for it in the EFM and also the behavior should be described within an autonomous functional feature state. In this case, this functional feature has a relation to the functional feature navigation: “navigation implies navigation setting”, which must be defined as a constraint in the EFM. The connections between the navigation functional feature and the navigation settings functional feature must also be described in the menu behavior SC, e.g. with inter-function transitions (Section 7.2.1). Alternatively, if the navigation settings service is regarded as a sub-function of the navigation functional feature, then no functional feature must be created in the EFM. The behavior of the navigation settings function should be described inside the functional feature state of the navigation functional feature, e.g. within one or-state. In practice, such decisions must be made frequently. For instance, new services must often be added to an existing function, e.g. the “parking assist system” is provided as an additional service of the already existing assist function. The advantage of specifying the menu behavior of such a new service within an autonomous functional feature state is that the changes and the relations to the existing function can be clearly presented. Independently of the form in which a service should be realized, the main point is that for each functional feature state, there must also be a functional feature created in the EFM.

Inter-function transitions

Different functional features have connections to each other, e.g. telephone and navigation. If a contact with an address is available in the functional feature telephone, the user has the possibility to choose this address and start the navigation guidance directly from the telephone context. From the functional feature navigation, the user has the possibility to switch to the functional feature telephone in order to choose an available address as navigation goal. In order to describe this kind of inter-function activity in the menu behavior SC, inter-function transitions are defined:

Definition. *An **inter-function transition** is a transition if_tr = (n, s_{source}, e, c, a, s_{target}) where s_{source} and $s_{target} \in$ FStates with $s_{source} \neq s_{target}$ are two different functional feature states. The definitions of the remaining elements are identical as for a normal transition introduced in Section 5.2.*

The set of inter-function transitions is defined as InterFunctionTR $\subset$ TR of the SC*.

Figure 7.3 displays three inter-function transitions: the transitions between ‘NavigationRoot’ and ‘TelephoneRoot’ and the one from ‘RadioRoot’ to ‘MediaRoot’. We say these functional features are dependent on each other.

An inter-function transition is usually a screen-dependent transition as introduced in Section 5.2. Inter-function activities are usually not intended or allowed from each screen of a functional feature. Even so, the inter-function transition is defined to directly start from and point to a functional feature state due to the impossibility of specifying all view states of these two functional features inside one diagram. The semantics of an inter-functional transition if_tr = (n, s_{source}, e, c, a, s_{target}) which starts from $s_{source} = [n, (s_0, ..., s_p), l, T]$ and points to s_{target} = [n’,(s'_0 ,...,s'_q),l’, T’] is a set of transitions $(if\_tr_0, ..., if\_tr_n)$ where n $\leq$ p, source(if_tr_i) = s_i and target(if_tr_i) = s'_j for $0 \geq i \geq n$ and $0 \geq j \geq q$.

Inter-function activities which are described with inter-function transitions are distinguished from “feature interactions”. Feature interactions occur when one feature modifies the behavior of another feature, whereas an inter-function activity in this thesis denotes the situation that results in a switch from one functional feature to another one. The thesis [13] addresses the problem of detecting conflicts and errors caused by feature interactions.

Breaking and broken states

Definition. *A simple state s is a* ***breaking-function state*** *if s is the source state of a transition that is in the semantics of inter-function transition.*

Inter-function transitions start and point to functional feature states because it is impossible directly to connect a sub-state of a functional feature state with a sub-state in another functional feature state due to the diagram concept. A breaking-function state is a state from which an inter-function transition should start. It is called a breaking-function state because this state offers the possibility to break out of the current function and access another function. A breaking-function state is usually a view state, since usually a screen offers the possibility to switch to another function.

Definition. *A simple state s is a* ***broken-function state*** *if s is the target state of a transition that is in the semantics of inter-function transition.*

A broken-function state is a state to which an inter-function transition should point. It is called a broken-function state because the current function is “broken in” by an inter-function transition from this state. A broken-function state can be a view state and also a pseudo state [3] such as a history state [69] [3]. Also, a breaking-function state can be a broken-function state and vice versa.

For example, the transition which is labeled with the event ‘confirm’ in Figure 7.3 allows the user to switch directly from the functional feature navigation into telephone in order to choose an existing contact address as a destination. This option is only provided in the view state ‘NavToAdrbook’ (shown in Figure 6.1) by pressing a contained button which triggers the event ‘confirm’. The view state is a breaking-function state since it offers the possibility to break out of the current functional feature. The transition with the event ‘confirm’ leads the user to the view state ‘TelContact’ in which existing contacts are listed. So ‘TelContact’ is a broken-function state since the current functional feature telephone is broken in this state.

7.2.2. How to incorporate variant features into the menu behavior SC

Variant features affect the allowed functional features and functional feature options of an HMI product, as already introduced for the product line shown in Figure 7.2. The variant features also impact the look that is the representation of screens of an HMI product. For instance, the screen enabling the user to define a navigation goal must additionally display an entry “province” for the Chinese market. This will be introduced later in Section 7.3. In this section, it will be introduced how variant features affect the menu behavior of a product. For instance, the menu behavior for the Chinese market must allow users to input a province. In this section, we introduce how variability caused by variant features can be described in the menu behavior SC.

In [103], the authors use variation points and junction points in their test models, which are activity diagrams, in order to describe the allowed activity sequences for different products of the PL. We use the same idea in our work in order to describe variability in the menu behavior which is caused by variant features.

Definition. *A **variation point** is defined as a pseudo state: vp = [jp,* $(v_0, ..., v_k)$*,* $(t_0, ..., t_k)$*] ∈ VPoints of SC*, where jp ∈ JPoints is the associated joining point;* $v_0, ..., v_k$ *is a set of variant features ⊂ VFeatures under the same node (or a set of functional feature options ⊂ FFOptions(ff) with ff ∈ FFeatures);* $t_0, ..., t_k \subset TR$ *are the outgoing transitions from jp,* t_i *and the following menu behavior describe the menu behavior of* v_i *for* $1 \geq i \geq k$

Definition. *A **joining point** is defined as a pseudo state* $jp = [vp, (tr_0, ..., tr_k)] \in$ *JPoints of SC* where vp ∈ VPoints is the associated variation point and* $tr_0, ..., tr_k \subset TR$ *are the incoming transitions of jp. The separate descriptions of the menu behaviors of* $v_0, ..., v_k$ *from vp must be joined via the transitions* $tr_0,...,tr_k$ *at jp.*

Figure 7.4 shows the graphical notations used for variation point and joining point.

Figure 7.4.: Graphical notations for variation point and joining point

Figure 7.5 shows a menu behavior SC, in which the variation and joining points are used to describe variability caused by the market variants EU and CH, as defined in Figure 7.2. The menu behavior SC describes how a navigation goal can be defined respectively for the European and Chinese market. Basically, the menu behavior for the European market is the same as shown in Figure 5.8 where variability was not considered. For the Chinese market, the menu behavior is defined in order to allow users to input a province. From the point at which the destination goal has been completely defined, the joining point defines that the menu behaviors for both markets are identical again.

There can be parts in the menu behavior of the HMI which are not affected by feature variants. For instance, the menu behavior of the bluetooth functional feature is independent of market variants or any other variant features.

In practice there can be situations in which nested variation and joining points are needed. For instance, as demonstrated in Figure 7.6, the menu behaviors of the Chinese and Japanese markets deviate first from the menu behavior of the European market. Then the menu behaviors of the Chinese and Japanese markets deviate from each other at a later point. Or for instance, as shown in Figure 7.7, one variation and joining points pair is used to specify variability caused by the market variant and another pair is used to specify variability caused by the system hardware variant. In the next section, it will be introduced that variation and joining points are also used to describe variability caused by functional feature options. Nested variation and joining point pairs can also be necessary.

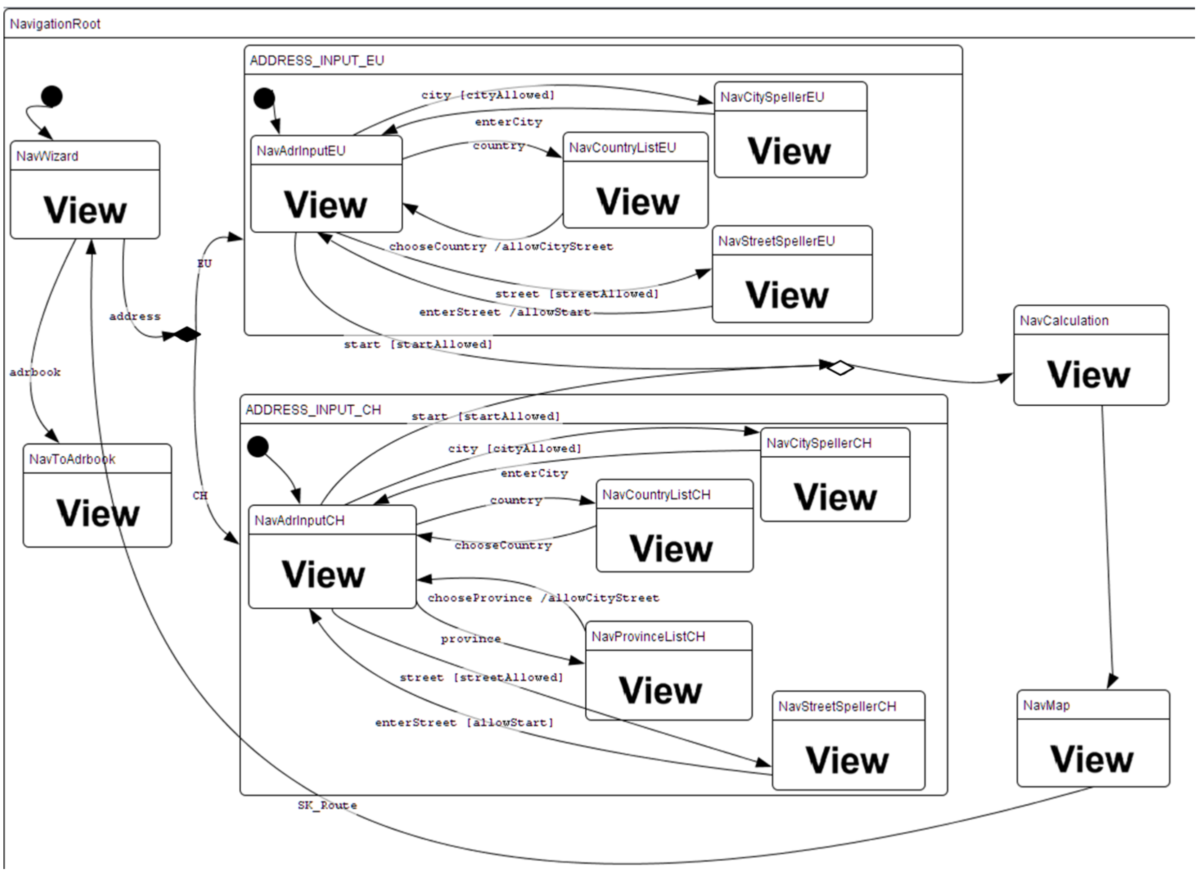


Figure 7.5.: Variation and joining points

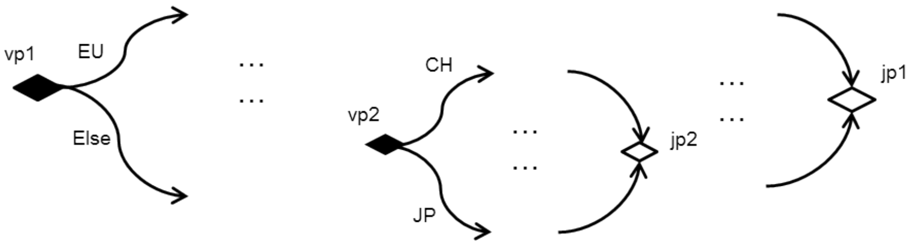


Figure 7.6.: Nested variation and joining points case 1

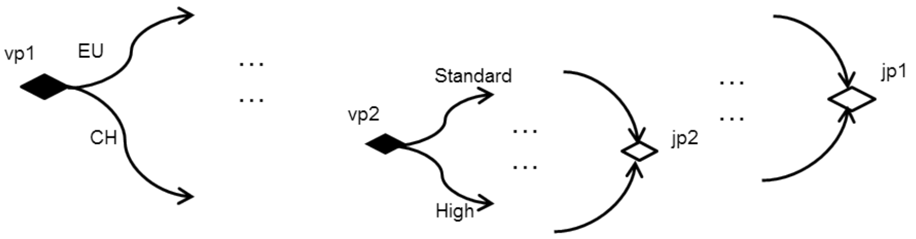


Figure 7.7.: Nested variation and joining points case 2

7.2.3. How to incorporate functional feature options into the menu behavior SC

In an infotainment system, the same functional feature can often be provided with different options. In the sample product line shown in Figure 7.2, the functional feature telephone can be provided with the options "BTHandfree" or "comfort". Functional feature options of the same functional feature differ in the menu behavior; however, they also share a large set of commonalities. For instance, in practice, the functional feature navigation can be provided in "NavigationHigh" or "NavigationStandard". About 95% of their menu behaviors are identical (Section 9.1.2). Therefore, we propose to describe the menu behavior of a functional feature within a functional feature state as introduced in Section 7.2.1 and apply variation and joining points to describe the differences of available options inside the functional feature state.

For the variation point, it has been already defined in the last subsection that $v_0, ..., v_k$ of a variation point vp = [jp, $(v_0, ..., v_k)$, $(t_0, ..., t_k)$] can be a set of functional feature options of a functional feature. The usage of variation and pointing points for functional feature options is similar to that for variant features. Figure 7.8 abstractly shows how variation and joining points are used to specify the menu behavior differences of options "BTHandfree" and "comfort" inside one functional feature state.

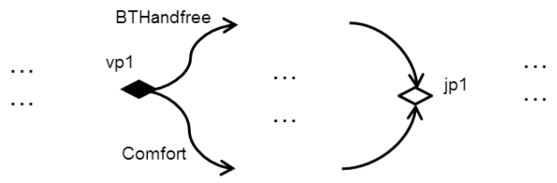


Figure 7.8.: Use of variation and joining points to specify variability caused by functional feature options

7.2.4. Related work

In this chapter, it was introduced how variability can be described in the menu behavior SC. In our approach, both the common behavior and the product-specific behavior are described within one SC. They can be distinguished with the help of functional feature states, variation points and joining points. In this section, we will introduce related research efforts which focus on the description of variability in SC diagrams or some other (behavior) diagrams. They use different methods or elements to distinguish between the common and the product-specific behavior.

In [44], as in our approach, both the common and specific behaviors are described in **one single SC**. The difference to our approach is that different types of SC elements are used to distinguish between the common and the product specific behavior: **optional state** and **optional transition**. We have decided against this approach. First, this information is duplicated. The information of whether the menu behavior which is described within a state is optional for a product is not, alone, enough for the code or test generation. A feature model must be, and is, also used in this approach to define the product line, as well as *product* *configurations.* If any changes occur in the product line, e.g. an optional feature becomes a standard feature, both the feature model and the menu behavior SC must be changed. Secondly, in order to define whether a state or a transition is optional, the HMI specifier has not only to possess knowledge about the behavior of the functional feature, but also inform himself about the current product line definitions. In our approach, any product line information is centrally managed within the feature model and feature model configurations.

In [120], state machines are left unchanged. Only the content classes of state machines are changed. Instead of using **inheritance** between state machines, inheritance between their context classes is used. The state machines are used as a behavior description of these classes. In this way, a product of an inheriting class contains not only the feature which is described in its own state machine, but also the features which are described in the states machines of the inherited classes. The first problem is the difficulty of specifying the connections between different state machines. Second, if the features of one product are changed, all inheriting products which are not intended to be changed must be redefined. Furthermore, the concept of a feature model is much easier and more intuitive than content classes of state machines.

In contrast to our approach, the approach in [114] describes each feature of the product line in a separate SC. This means that each product is described with **several SCs** (a subset of these SCs) and their relationships to each other. Each feature in the feature model is associated with a SC. The SCs which together should describe the behavior of a particular product can be derived from the respective *configuration.* Therefore, the challenge is to bind several SCs together instead of distinguishing between the common and the product-specific behavior in one single SC. Basically, it is not critical whether to use one single SC or several SCs. We have decided on one single SC since the HMI specifications and development models in practice are usually based on one single SC in which some composite states are used to describe the behavior of different features. This is more easily acceptable.

Specifying variability which results from different variant features and functional feature options is not within the focus of any of the three introduced approaches.

In [103], **activity diagrams** are used as test models which describe the allowed user action sequences in the domain engineering phase. *Variation points* and *junction points* are used to describe how different products of the product line can differ in the user action sequences. "*A junction point is a special node in the test model in which all variants of the variation point are joined and continue in the same control flow.*" In the subsequent work [113], the activity diagram is called a ***flow-based test model*** and is used to describe the data flow of the system. The variation and junction points are used in a very similar way as in our approach. The small difference is that each outgoing transition of a variation point represents a complete product of the product line. Complex feature combinations or options are not in the focus of their work.

Besides the dynamic behavior, variability can also occur in other aspects of a software system. For instance, in [74], [56] and [48], **use case models** are extended for variability. In [24] and [22], different **UML diagrams** are extended for variability.

7.3. Representation layer with variability

In Section 5.3, the meta-model for the representation layer was introduced. Figure 7.9 shows the meta-model presented in Figure 5.15 again.

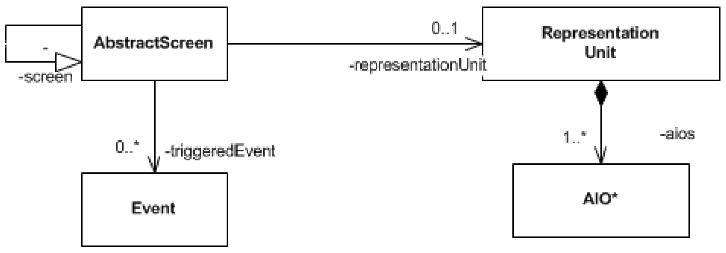


Figure 7.9.: Meta-model for the representation layer without variability

The meta-model above defines elements which can be used in the representation layer to specify the screens. Here, variability has not yet been taken into account. In this section, the representation layer will be extended for variability by adding parameterized inheritances into the meta-model.

We use parameterized inheritances [39] [109] in order to describe variability in the representation layer. A parameterized inheritance is an inheritance of which the super-class is specialized by parameters. Up to now, normal inheritance has been used in the meta-model in order to simplify the specification of possible screen-triggerable events. In contrast, parameterized inheritances will be used to specify variability in the representation layer.

7.3.1. Variability of abstract screens

Sometimes functional features, functional feature options, and variant features can affect the events which can be triggered by a screen. In this section, we introduce how variability can be described for abstract screens.

The functional features provided in a particular product affect the triggerable events of some screens. For example, the screen ‘Main’ (shown in Figure 2.4) must only trigger the event ‘nav’ if the functional feature navigation is available in the product. Screen ‘Main’ also has a basis event - the radio event - which can be triggered in any product of the PL. Therefore, in order to avoid repeated specifications of such basis events for each provided product, parameterized inheritances can be used. Figure 7.10 shows how parameterized inheritance is used for the screen ‘Main’.

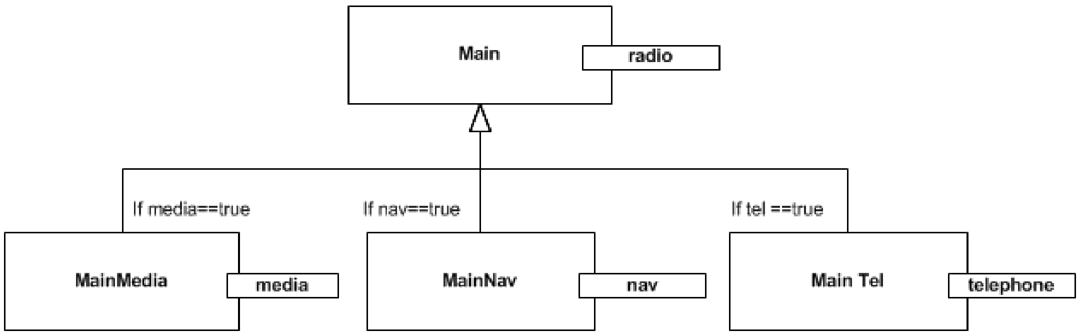


Figure 7.10.: Inheritance of an abstract screen caused by functional features

The parameterized inheritances in Figure 7.10 have the following meaning: first, the 'radio' event can be triggered in each product of the PL. Second, if the functional feature media is available in a product, then the 'media' event can be triggered from 'Main'. Likewise, if the functional feature navigation or telephone is available, then the 'nav' or 'telephone' event can be triggered. Third, if two or all three of the optional functional features are available in a product, then the respective two or three events can be triggered. That means the inheritances do not mutually exclude each other. If an abstract screen has no inheriting screens, then its triggerable events are independent of the provided products.

Similarly, triggerable events of a screen can be dependent on the options in which functional features are provided and the variant features.

Figure 7.11 shows an example of how variability caused by the market variant can be described via parameterized inheritance:

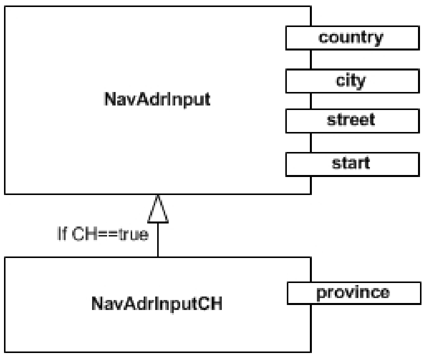


Figure 7.11.: Inheritance of an abstract screen caused by variant features

Screen 'NavAdrInput' shown in Figure 2.5 can basically trigger the 'country', 'city', 'street' and 'start' events for the available markets. For the Chinese market, it must additionally trigger the 'province' event which enables the input of a province.

If inheritances exist, view states in the menu behavior layer are associated with the inherited abstract screens. Triggerable events for an individual product must be calculated based on the parameters of the inheritances. For instance, both the view states 'NavAddressInputEU' and 'NavAddressInputCH' presented in Figure 7.5 are associated with the inherited abstract screen.

7.3.2. Variability of representation units

Functional features, functional feature options and variant features also affect the contents of some screens and hence cause variability in the representation units. In this section, we introduce how parameterized inheritances can be used for representation units in order to specify variability.

The functional features provided in a product are always provided to the user via screens. For instance, if the functional feature navigation is available, the wizard entry with the text "navigation" should be presented in screen 'Main' as shown in Figure 2.4. All other UI elements, e.g. the title, soft keys etc., are always displayed independently of the provided functional features. In order to avoid repeated specifications of these basic elements, parameterized inheritances are used, as shown in Figure 7.12.

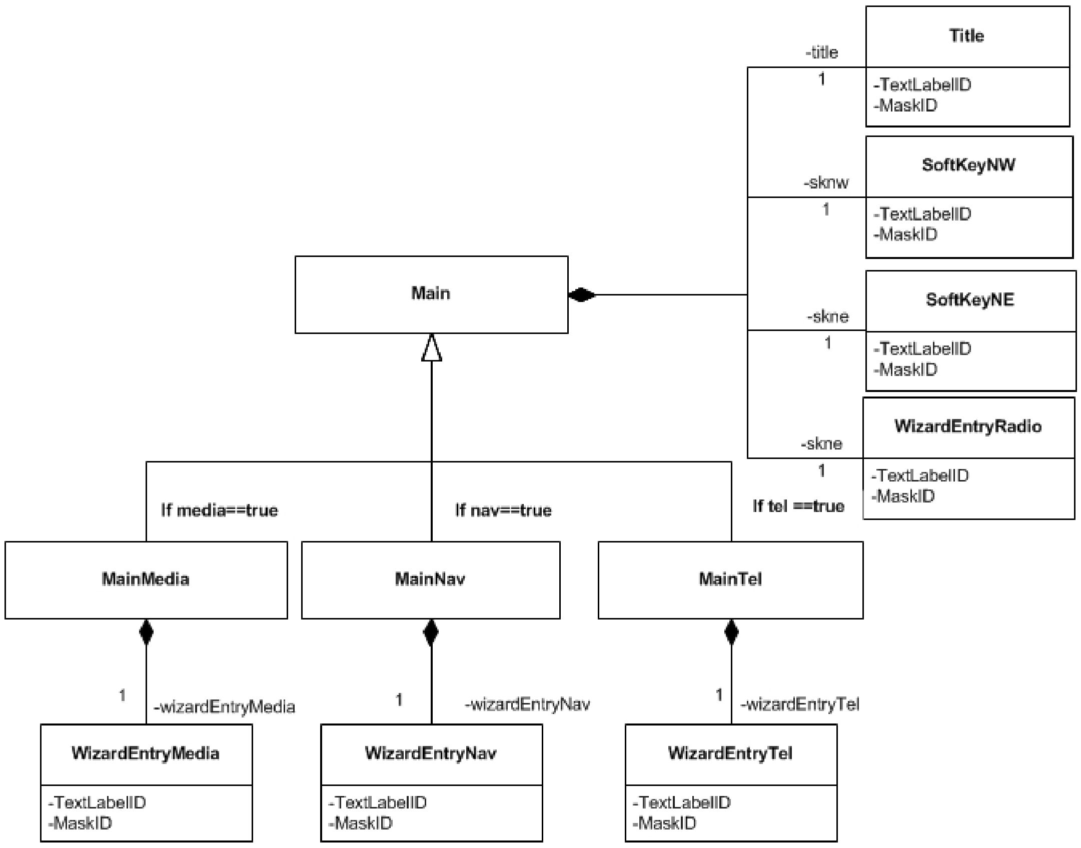


Figure 7.12.: Inheritance of the representation unit 'Main' caused by functional features

As explained in the last subsection, the parameterized inheritances used here do not mutually exclude each other. That means if a product provides, e.g. both the telephone and navigation functional features, both wizard entries must be displayed by the screen 'Main'.

Similar to the functional features, functional features options and variant features also have an influence on the contents of some screens. Figure 7.13 shows how variability caused by the market variant can be described via parameterized inheritances.

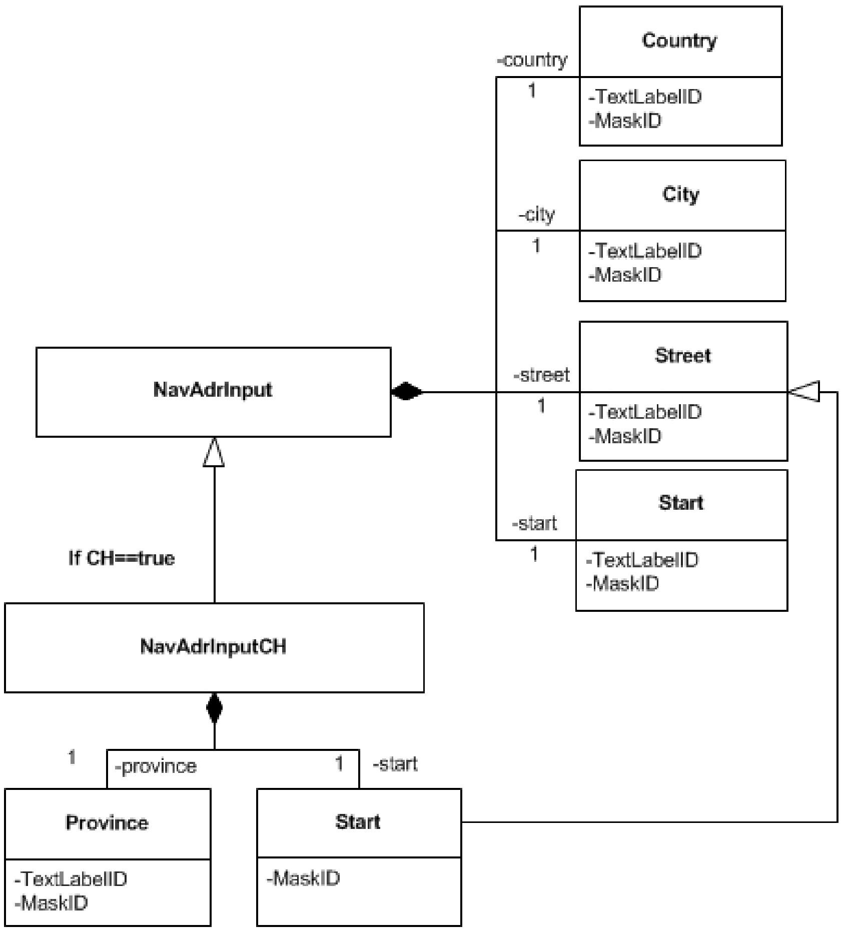


Figure 7.13.: Inheritance of the representation unit ‘NavAdrInput’ caused by variant features

The diagram above defines that if the market variant of the product to be tested is the Chinese market, then the ‘province’ widget should be displayed in addition to the ‘country’, ‘city’, ‘street’ and ‘start’ widgets. Furthermore, the ‘start’ widget gains a new position due to the additional widget before it. The old position is overwritten in the inheriting class.

7.3.3. Extended meta-model for the representation layer

According to the approaches describing variability introduced above, the meta-model shown in Figure 7.9 is extended as follows:

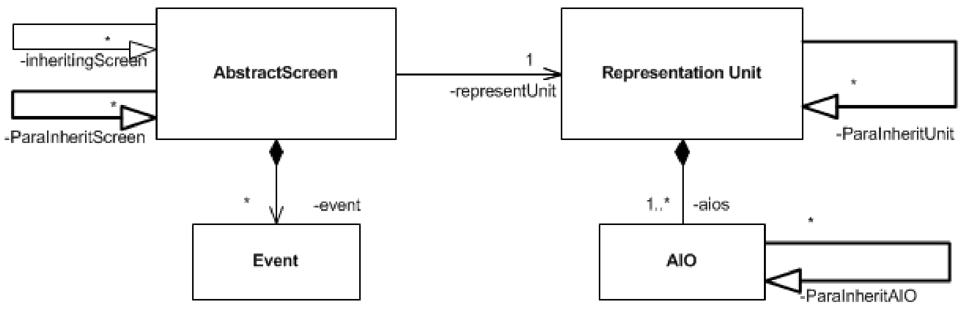


Figure 7.14.: Extended meta-model describing variability for the representation layer

The bold generalization arrows in the diagram above represent parameterized inheritances.

Figure 7.15 shows an instance of the meta-model which specifies the screen 'NavAdrInput'.

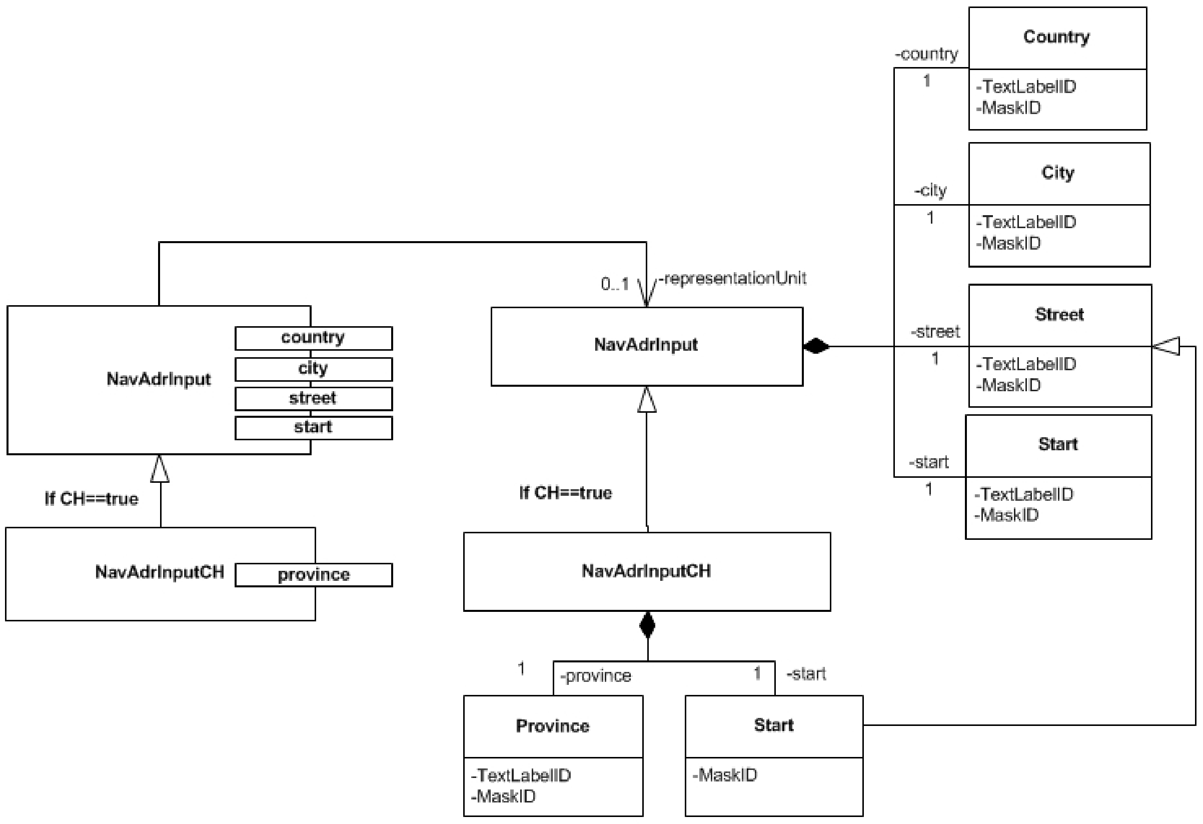


Figure 7.15.: An instance of the meta-model describing variability

In the representation layer, the triggerable events and the contained UI elements of the screens are specified. In this section, it was introduced how variations in the triggerable events and the contained UI elements between screens of different variants can be described via parameterized inheritances in the representation layer.

Variability can also exist in the design layer and the text layer of the test-oriented HMI specification. In practice, an HMI product line can also have different system variants, such as the "high" and "standard" systems. HMI products in the "high" variant have bigger displays and bigger screens than the products in "standard" variants. The UI elements of "high" products have different masks compared to those same UI elements in "standard" products. In this way, system variants result in variability in the design which is specified in the design layer in our approach. System variants can also result in variations in some texts. These must be specified in the text layer. In order to specify variabilities in the design and the text layers, a very simple method can be applied: work with two separate lists of masks or texts for the respective two variants. During the test instantiation, the parameters of the inheritances in the representation layer define which list should be accessed in order to obtain the values of the masks and the texts.

7.4. Related work

In this chapter, we have introduced how variability can be extended into a test-oriented HMI specification based on product line approaches. The basic approach is feature modeling combined with a behavior description using UML SC. A feature model defines the commonalities and variabilities of the PL on a high level, and a UML SC concretely defines these commonalities and variabilities on the behavior level. Related work has been separately introduced for feature modeling and modeling variability in behavior models. In this section, we introduce an alternative approach to our approach which is used in the domain of software product line development.

As introduced in Section 7.1.2 and Section 7.2.4, the approaches in [44], [114] and [120] are also based on feature modeling and describing variability in UML diagrams. An alternative approach which is based on the **aspect-oriented** software development is available for product line development.

The core problem in the software product line is to model variability. The relationship between the commonality and variability here is quite similar to the relationship between crosscutting concerns and core concerns in aspect-oriented modeling. Therefore, it is possible to model a product line with aspect-oriented techniques. In [111], an approach has been introduced which demonstrates how a product line can be modeled with the specification language *ADORA* based on aspect-oriented modeling; i.e. an abstract object should be created for a system. It describes the aspects of the system as sub-components. These aspects correspond to the features in the feature modeling concept. Different variants of an aspect can be separately described with SCs and so-called scenario-SCs and gain relationships to each other. The relationships used are quite similar to those in a feature model. Logical items including variables are used to describe a *join relationship* which corresponds to feature configuration in the feature modeling concept. Also the concept of constraints is used. The automatic derivation of an application (a variant) is based on available tools allowing automatic derivation of applications from models based on ADORA. In [117] an approach has been presented that allows variability implementation and management by integrating model-driven and aspect-oriented software development. Features are separately described in models and these models are composed using aspect-oriented composition techniques. In [5], aspect-oriented use case modeling is adapted for product lines. A study of how product line modeling can be supported by aspect-oriented methods can be found in [92].

In comparison to the aspect-oriented techniques, feature models and UML SCs are very intuitive and have already been accepted by the industry for practical applications. In particular, it must be considered that the people creating HMI specifications usually do not have a technical background. Aspect-oriented modeling techniques are “oversized” for the goal of creating HMI specifications.

7.5. Conclusions

Variability can be caused first by the functional features provided by a product, second by the options in which functional features are provided, and third by the variant features of the product. In this chapter, we have introduced how the variability caused by these factors can be integrated into test-oriented HMI specification.

Firstly, an extended feature model (EFM) was proposed, which is used to describe an HMI product line. Extended feature configurations (EFMConfig) respect the respective EFM and describe single HMI products. Secondly, new elements are inserted into the SC in order to describe variability. A SC describing variability is defined as a SC*. In a SC*, the behaviors of different functional features are separately specified with the help of functional feature states. Finally, variability is also integrated into the representation layer with the help of parameterized inheritances.

By doing this, the test-oriented HMI specification provides the basis for testing an HMI product line and its HMI products.

Chapter 8.

Test Generation With Variability

In the last chapter, a test-oriented HMI specification was introduced which describes the expectation of all potential products of an HMI product line. In this section, we focus on the problem of how menu behavior tests can be generated from such a test-oriented HMI specification taking into account variability in the different products to be tested.

The largest challenge here is to avoid redundant test generations. As explained previously, different HMI products of the same product line share a large set of menu behaviors. If the test generation activity must be repeated for each product to be tested, the common set of menu behaviors will be requested as often as the number of products to be tested. Due to the high degree of commonality and the large number of products to be tested in practice, much generation time would be wasted. Furthermore, the generated tests for different products would have a large set of identical tests or test steps which verify the common behavior of different products. Redundant executions of these tests would also lead to a heavy loss of time.

In this section, we introduce a test generation method which first generates tests for all potential products to be tested, second, avoids redundant test generations, and third, allows us to identify common tests and common test steps in order to preserve them from redundant executions.

The test generation based on the transition coverage will first be introduced in detail. Later in Section 8.2, test generation based on some other coverage criteria will be discussed.

8.1. A test generation method

In Section 6.2, a test generation algorithm based on transition coverage was introduced, which, however, does not consider variability of HMIs. In this section, we first prepare the menu behavior SC for test generation and extend this algorithm to generate test sequences which are parts of tests for all potential products to be tested (Subsection 8.1.1), and then introduce how tests can be created from these sequences for a particular product to be tested (Subsection 8.1.2).

Test sequences are a sequence of test steps and parts of tests. Tests can be created from several test sequences. The formal definition of a test sequence will be introduced later.

Figure 8.1 shows again the three products provided by the product line which was introduced in the last chapter and shown in Figure 7.2. The demonstration of the test generation strategy is based on the assumption that tests should be generated for Product 2 and Product 3.

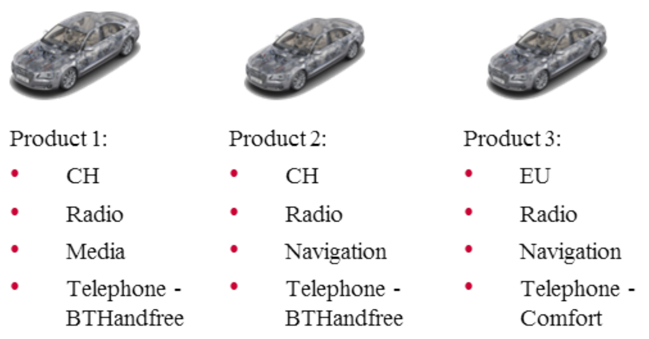


Figure 8.1.: Three products of a product line

8.1.1. Generating test sequences

The idea of the test generation is that the test generation algorithm only traverses the menu behavior SC once and generates all test sequences which are potentially needed for any provided product of the PL. Tests should then be created from these sequences for a particular product to be tested in a very efficient way.

Step 1 Release and preserve dependencies for each relevant functional feature state

A generated test sequence can remain within one functional feature, as in Test 1 below which remains inside the functional feature navigation. A test sequence can also "cross" different functional features, as Test 2 does from navigation to telephone.
Test 1:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap]

Test 2:
[Main] → (nav) → [NavWizard] → (adrbook) → (NavToAdrbook) → [TelContact] → ...

Different products provide different combinations of functional features. If generated test sequences cross several functional features, as in Test 2 from navigation to telephone, they can only be performed on some of the products, e.g. Products 2 and 3, but not on other products, e.g. Product 1. Therefore, in order to generate test sequences which can be used later to create tests for different products, generated test sequences should remain within one functional feature. This can only be realized if each functional feature state is considered as a separate entity by the generation algorithm. Functional feature states have different kinds of dependencies. Therefore, these dependencies should be first released and second preserved against data loss, since they may be used later to fulfill the chosen coverage criteria.

Step 1.1 Ignore irrelevant functional feature states and their transitions
Assume that the products to be tested are described with EFMConfs and FF is the set of relevant functional features of these products to be tested. For each functional feature state $ff\_state \in$ FStates: if FunctionalFeature(ff_state) $\notin$ FF then ignore ff_state and its incoming and outgoing transitions

This step is relevant if only a subset of the products provided by the PL are to be tested. It avoids the generation of tests from functional features which are not embedded in the products to be tested.

Figure 8.2 shows the input SC for this step. For products 2 and 3, the functional feature media is irrelevant. The functional feature state 'MediaRoot', the global transition pointing to it, and the inter-function transition originating from it should be ignored for the test generation. Ignoring these transitions does not lead to data loss, since if a functional feature is not provided in a product, there are no global actions leading to it, and no inter-function actions between it and other functional features. The result of this step is shown in Figure 8.3.

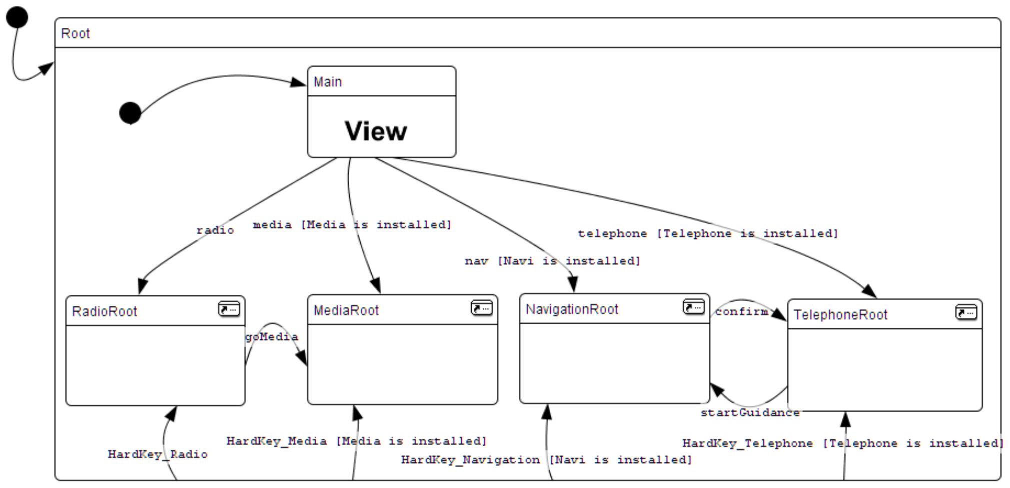

Root
Main
View
radio
media [Media is installed]
nav [Navi is installed]
telephone [Telephone is installed]
RadioRoot
MediaRoot
NavigationRoot
TelephoneRoot
goMedia
confirm
startGuidance
HardKey_Radio
HardKey_Media [Media is installed]
HardKey_Navigation [Navi is installed]
HardKey_Telephone [Telephone is installed]

Figure 8.2.: Root state of the SC for test generation

Step 1.2 Release and preserve dependencies on global transitions for each relevant functional feature state
Assume state $s_{root} = [n, (s_0, ..., s_k), l, T]$ is the result of the last step. For each semantical transition of each global transition global_tr $\in$ GlobalTR (defined in Section 5.2) create a global transition entry globalTr_entry = [e, target, cons] and ignore the global transitions for the generation algorithm

In Section 7.2.1, it was explained that global transitions realize actions which are effective at any time and in any system state. The semantics of global transitions mean that they can be started from each view state of the SC. A test generation that considers global transitions would lead to a test explosion. Furthermore, it is not necessary to test every global transition from every system state. Usually, if a global transition can be triggered from one or some of the states, it can also be triggered from other states. Although there is no guarantee of this, to assume it is a good trade-off between the testing effort and testing quality. For this reason,

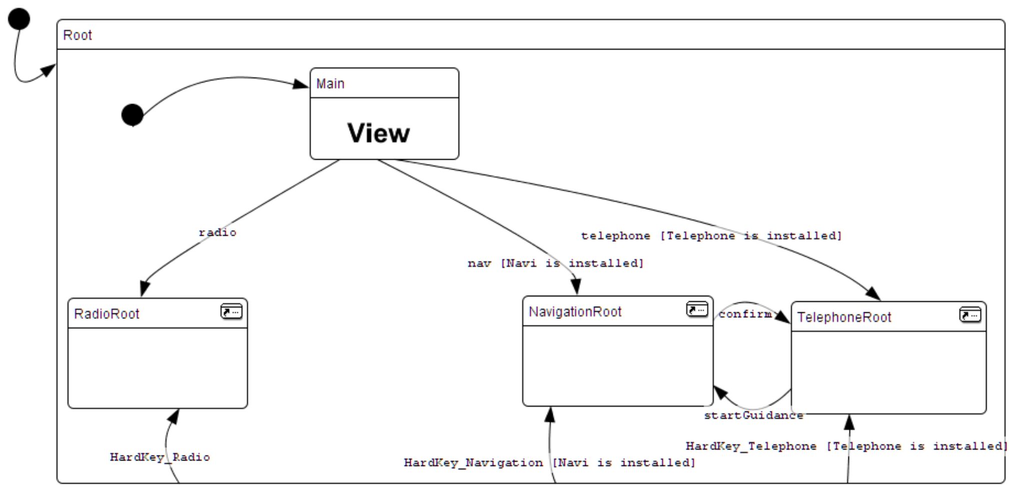

Root
Main
View
radio
telephone [Telephone is installed]
nav [Navi is installed]
RadioRoot
NavigationRoot
confirm
TelephoneRoot
startGuidance
HardKey_Telephone [Telephone is installed]
HardKey_Radio
HardKey_Navigation [Navi is installed]

Figure 8.3.: Result of ignoring irrelevant functional feature states and their transitions

global transitions should be ignored by the test generation for the present and preserved for later usage. This ensures that no data is lost for fulfilling the coverage criteria.

Figure 8.4 shows the result of ignoring the global transitions for the test generation.

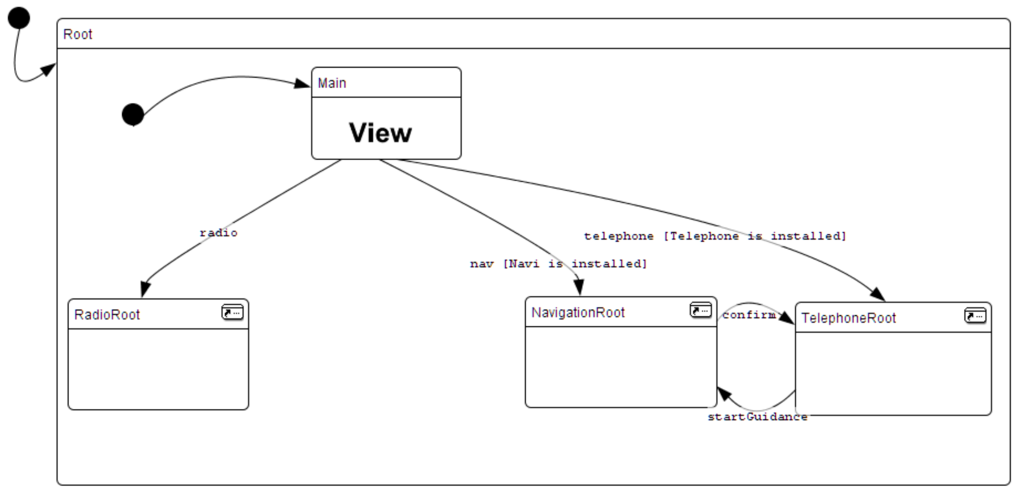


Figure 8.4.: Result of ignoring global transitions

For each global transition, a global transition entry should be created in order to preserve it.

Definition. *A **global transition entry** globalTr_entry for a global transition global_tr = (n , s_{source}, e, c, a, s_{target}) is defined as a tuple globalTr_entry = [event, target, cons] where event = e, target = s_{target} if s_{target} is a simple state, otherwise if s_{target} is an or-state, then target is the first simple state in s_{target}, cons = c.*

We use the following notation for a global transition entry: $\xrightarrow{\text{event [cons]}}$target.

The global transition entries created for the three ignored global transitions are:

$$\text{globalTr\_entry1} = \xrightarrow{\text{HardKey\_Navigation [Navi is installed]}} \text{NavWizard}$$

$$\text{globalTr\_entry2} = \{\xrightarrow{\text{HardKey\_Radio}} \text{RadioSenderList}\}$$

$$\text{globalTr\_entry3} = \{\xrightarrow{\text{HardKey\_Telephone [Telephone is installed]}} \text{TelephoneMain}\ \}$$

For the present, tests are generated without the global transitions. In Section 8.1.2, Step 3.1 will extend generated tests with global transition entries.

Step 1.3 Release and preserve dependencies between relevant functional feature states

Inter-function transitions represent dependencies between functional features. In order to generate test sequences that remain within one respective functional feature state, inter-function transitions must, like global transitions, be ignored for the present by the test generation and preserved for later usage.

An inter-function graph is used to preserve the dependencies between the functional features in a product. It is a common graph with the following characteristics:

Definition. *An **inter-function graph** can be defined as a tuple InterFGraph = (N, E) where N is the set of nodes and E is the set of the edges, a node n ∈ presents a unique functional feature: FunctionalFeature(n) ∈ FF, an edge e ∈ E presents a dependency between two functional features, SourceFunctionalFeature(e) = FunctionalFeature(source(e)) and Target-FunctionalFeature(e) = FunctionalFeature(target(e)), a node n ∈ N must not have an edge, each edge e is associated with a set of transition entries TransitionEntries(e).*

The semantics of an inter-function transition is a set of transitions, as introduced in 7.2.1. Each semantical inter-function transition is preserved via an inter-function transition entry:

Definition. *An **inter-function transition entry** of a semantical inter-function transition* $if\_tr = (n\ , s_{source}, e, c, a, s_{target})$ *is defined as a tuple intTransition_entry = [event, source, target, cons] where event = e, source =* s_{source}*, target =* s_{target} *and cons = c.*

We use the following notation for an inter-function transition entry: source $\xrightarrow{\text{event [cons]}}$ target, e.g. NavToAdrbook$\xrightarrow{\text{confirm}}$ TelContact.

Products 2 and 3, as shown in Figure 8.1, provide the same functional features and their inter-function graphs are identical. Figure 8.5 shows the inter-function graph and the inter-function transition entries.

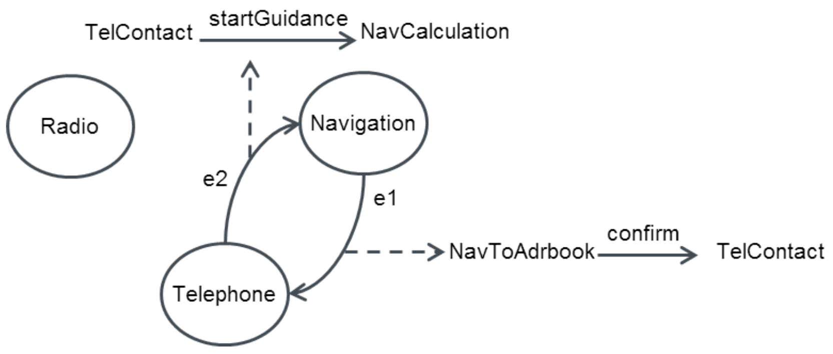


Figure 8.5.: An inter-function graph

Step 1.3 contains the following 4 sub-steps:

Step 1.3.1 *Create an inter-function graph for each product to be tested based on the following rules:*
1. For each functional feature ff ∈ FF of EFMConf, create a node n with FunctionalFeature(n) = ff in the inter-function graph
2. For each inter-function transition if_t = (n , ff_state_{source}, e, c, a, ff_state_{target}) ∈ TR: first insert an edge e with SourceFunctionalFeature(e) = ff_state_{source} and TargetFunctionalFeature(e) = ff_state_{target}, second, obtain the semantical transitions: Semantics(if_tr) = (tr_0, ..., tr_k), then for each tr_i with $0 \leq i \leq k$, create an inter-function transition entry transition_entry and add it into TransitionEntries(e)

The result of this step for Product 2 or Product 3 is shown in Figure 8.5.

Each edge e in an inter-function graph represents a dependency of the functional feature TargetFunctionalFeature(e) on the functional feature SourceFunctionalFeature(e). As shown in Figure 8.5, the functional feature telephone is dependent on the functional feature navigation via the edge e1, and navigation is dependent on telephone via the node e2.

Function. *For an inter-function graph, the function GetDependentTransitionEntries(ff ∈ FFeatures) provides for a given functional feature ff the set of inter-functional transition entries of edges starting from its node.*

This function obtains inter-function transition entries of inter-function transitions, which break out from the given functional feature and lead to other functional features.

Step 1.3.2 *Mark the breaking-function states and broken-function states as follows:*
For each inter-function transition, assume that Semantics(InterFunctionTR) = {tr_0, ..., tr_k}. For each tr_i with $0 \leq i \leq k$, mark the state source(tr_i) as a breaking-function state and the state target(tr_i) as a broken-function state

Breaking-function and broken-function states are important for the test sequence generation (see block 3.3 and 3.4 in the algorithm 8.1.1).

Step 1.3.3 *Ignore all inter-function transitions for the test generation*

The result of this step is shown in Figure 8.6.

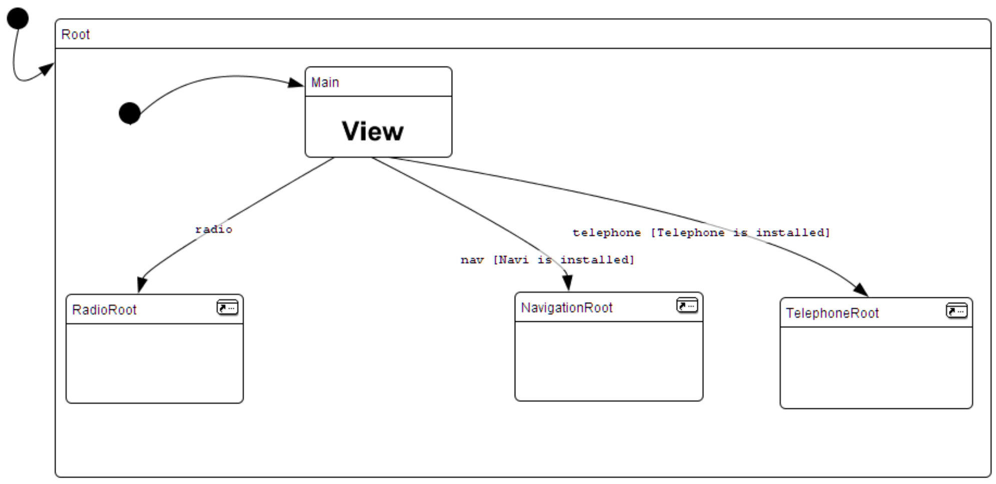


Figure 8.6.: Ignore inter-function transitions

Step 1.3.4 *For each functional feature state to be tested, ignore all other functional feature states and their transitions*

Figure 8.7 shows the result of this step for the functional feature navigation. For the functional feature navigation, this remaining SC content is the input for the test sequence generation algorithm, which is introduced as Step 2.

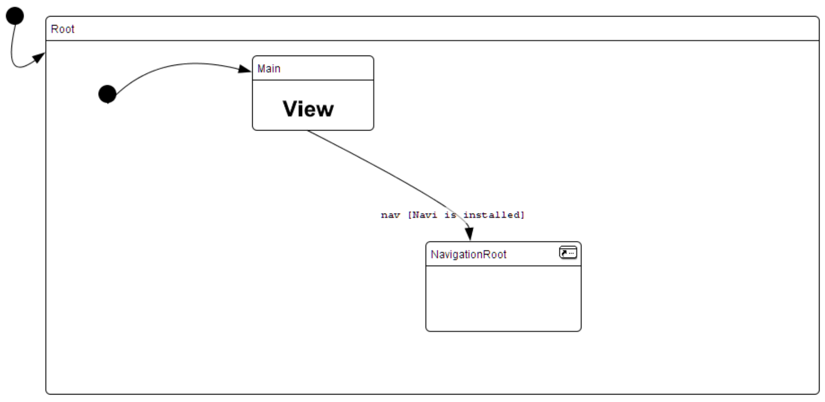


Figure 8.7.: Ignore other functional states

Definition. *We call the SC resulting from the above described steps a* ***released functional feature state for functional feature ff*** *where $ff \in FFeatures$. A released functional feature state contains an initial state and a functional feature state which is released from its dependencies on global transitions, inter-function transitions and other functional feature states*

Step 2 Generate test sequences for each relevant functional feature state

Until now, the preparation for the actual test generation has been done. For each product to be tested, an inter-function graph has been created. For each function feature to be tested, a released functional feature state is available for the test generation.

Step 2 generates test sequences for a particular functional feature from its released functional feature state. Parallel to the sequence generation, a so-called test sequence graph shall be created, which is used later for creating tests for a particular product from generated test sequences.

We demonstrate the test sequence generation based on the functional feature navigation. The released functional feature state for navigation has been shown in Figure 8.7. Figure 8.8 presents the flattened SC which has the same semantics as the hierarchical SC shown in Figure 8.7. At the beginning of Section 6.2, it was explained why flat SCs are preferred in this work for the test generation.

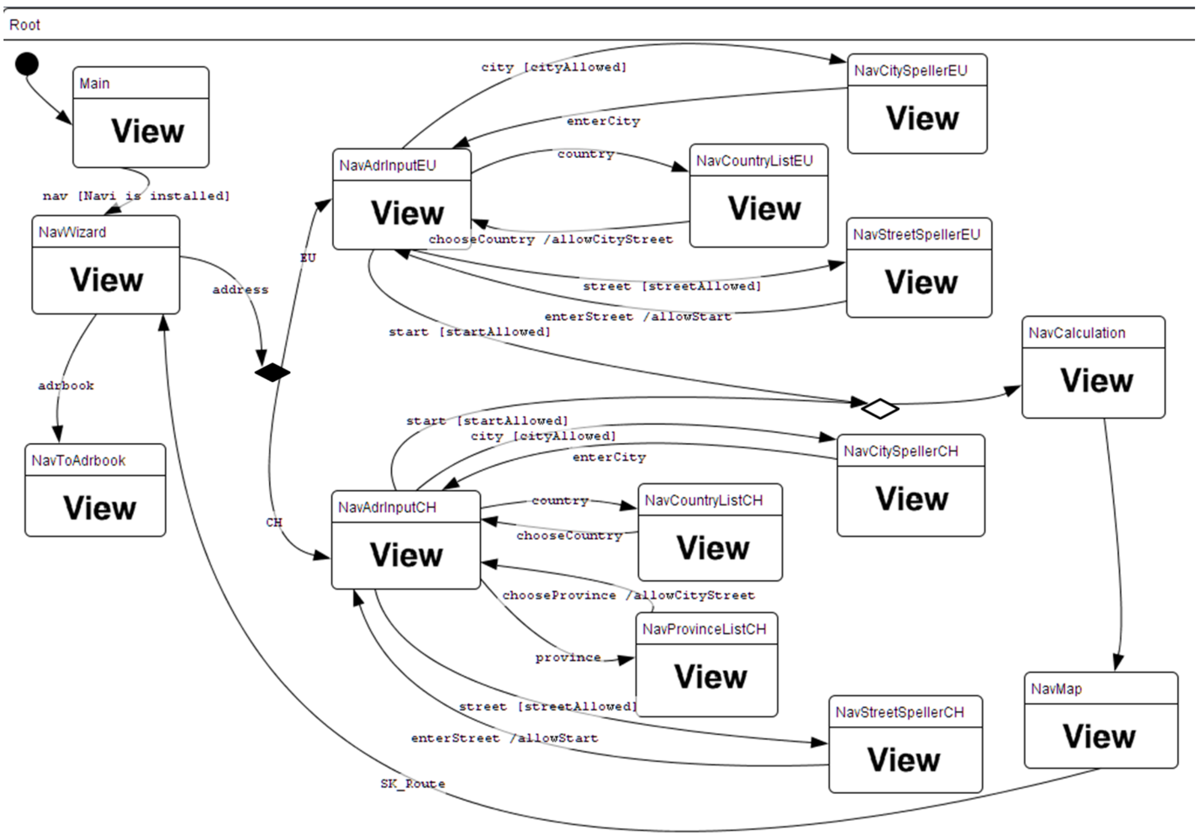


Figure 8.8.: Released functional feature state of functional feature navigation with a flat structure

First, the most important definitions will be introduced:

Definition. *A menu behavior test sequence, or* ***test sequence*** *in short, is defined as a tuple MTestSeq = (Cons, Steps) where Cons is a set of conditions and Steps is a sequence of test* *steps. The definitions of Cons and Steps are the same as in the definition of a menu behavior test 4.2.2. The difference is that $step_0$, which is the first step from Steps, can but does not have to be the name of the initial screen which is expected when the SUT is started.*

A test is composed of one or several test sequences in a particular order. A test starts from the initial screen and is executable if its conditions are fulfilled, whereas test sequences usually do not start from the initial screen and usually need other test sequences as previous sequences.

We use the same notation for test sequences as for tests.

Definition. *A **test sequence graph** seqGraph is a finite and directed graph without cycles. It is defined as seqGraph = (init, N, $VNodes$, $JNodes$, E) where "init" is the initial node, N is the set of nodes, each $n \in N$ is associated with a test sequence Seq(n), VNodes is the set of variation nodes, JNodes is the set of junction nodes and E is the set of the edges.*

Definition. *A **variation node** is defined as vn = [jn, $(v_0, ..., v_k)$, $(e_0, ..., e_k)$] $\in$ VNodes of a test sequence graph, where jn $\in$ JNodes is the associated junction node; $v_0, ..., v_k$ is a set of variant features $\subset$ VFeatures or a set of functional feature options $\subset$ FFOptions(ff) with ff $\in$ FFeatures, $e_0, ..., e_k \subset$ E are the edges outgoing from jn.*

Definition. *A **junction node** is defined as $jn = [vn] \in JNodes$, where vn $\in$ VNodes is the associated variation node in the test sequence graph.*

For variation nodes and junction nodes, we use the same notation as for variation points and joining points of a SC*.

Function. *For a node $n \in N$, the function **BreakingFunction(n)**= $\{true, false\}$ returns whether the associated test sequence Seq(n) ends with a breaking-function state.*

Function. *For a given inter-function graph, **GetNodesEndingWith(state s)** returns a set of nodes Nodes $\subset$ N where for each $n \in$ Nodes the associated test sequence Seq(n) ends with s.*

Function. *For a given inter-function graph, **GetNodesStartingWith(state s)** returns a set of nodes Nodes $\subset$ N where for each $n \in$ Nodes the associated test sequence Seq(n) starts with s.*

Basic functions available for a common graph such as parent(node n) and children(node n) are also valid for the test sequence graph.

An example of a test sequence graph can be found in Figure 8.10.

Now we are ready to introduce the test sequence generation algorithm. In order to simplify the introduction, we assume that nested variation and joining pairs have not occurred. We extend the algorithm described in Section 6.2 which is based on the transition coverage with the following new rules:

- The generation algorithm generates test sequences instead of tests.
- The generation algorithm generates a test sequence graph in addition to the test sequences.

- The generation algorithm takes variation points and joining points into account as follows:
  When the algorithm visits a variation point, it ends the current sequence, creates a node for it in the test sequence graph, and adds a variation node after the created node. The algorithm continues traversing the SC*. When it visits the respective joining point, current sequences are ended and the algorithm generates a junction node.

- The generation algorithm is extended to take account of breaking- and broken- function states as follows:
  When the algorithm visits a breaking-function state s, it ends the current sequence with s and creates a node for it. When the algorithm visits a broken-function state s, it ends the current sequence with the previous state of s, creates a node for it, and begins a new sequence starting with s.

This means that the algorithm cuts a test sequence at special elements such as variant points and breaking-function states. The goal is to break tests into test sequences at a good time so that sequences can be used later to construct tests for different products. The test sequence graph is used to manage the order of generated sequences. The input of the algorithm is a flattened released functional feature state of the particular functional feature to be tested. The algorithm should be performed for each functional feature to be tested.

Figure 8.9 graphically illustrates how the test sequence generation algorithm traverses the released functional feature state of navigation based on transition coverage.

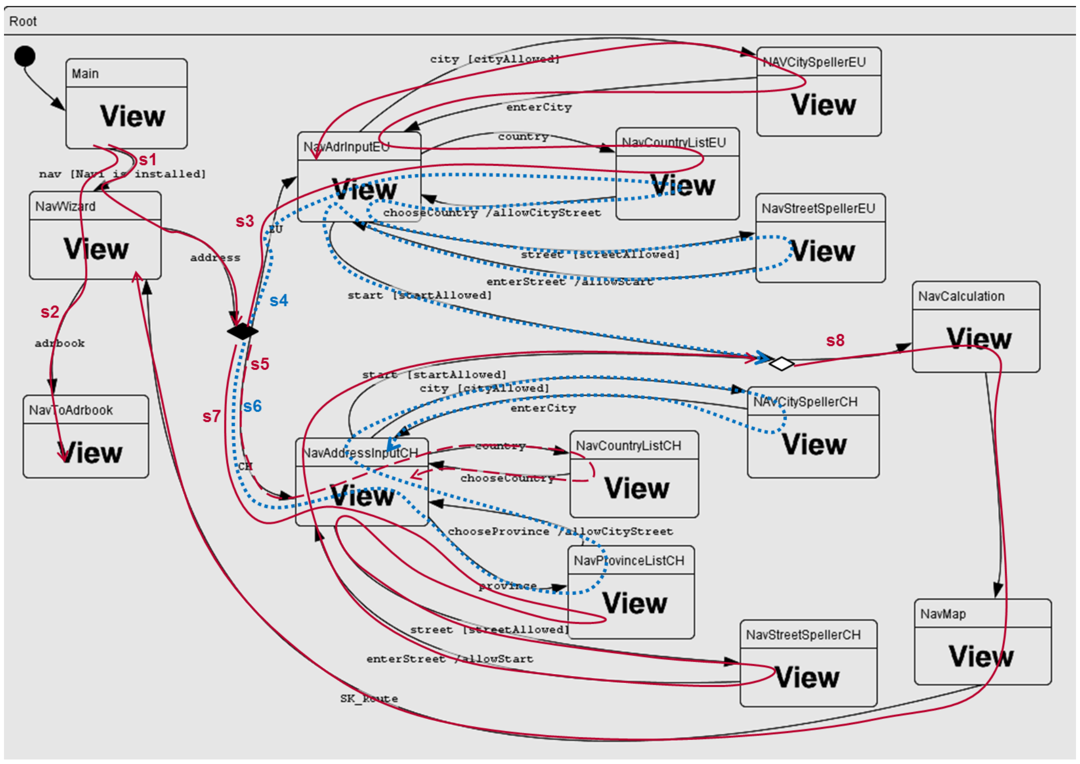


Figure 8.9.: Algorithm generating test sequences

The 8 generated test sequences are as follows:

s1:
[Main] → (nav) → [NavWizard] → (address)

s2:
[Main] → (nav) → [NavWizard] → (adrbook) → [NavToAdrbook]

s3:
[NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput]

s4:
[NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start)

s5:
[NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput]

s6:
[NavAdrInput] → (province) → [NavProvinceList] → (chooseProvince) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput]

s7:
[NavAdrInput] → (province) → [NavProvinceList] → (chooseProvince) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start)

s8:
[NavCalculation] → [NavMap] → (SK_Route) → [NavWizard]

Sequences s3, s4, s5, s6 and s7 are variant-feature specific. Sequences s1, s2 and s8 are variant-feature common.

The created test sequence graph is shown in Figure 8.10.

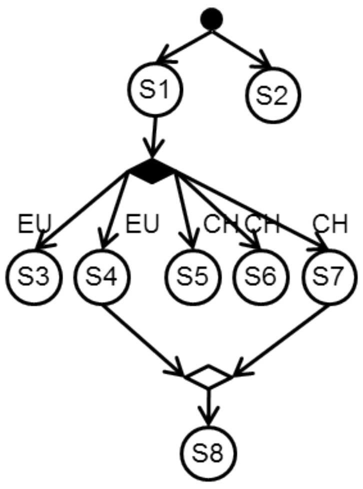


Figure 8.10.: Test sequence graph of functional feature navigation

The test generation algorithm breaks the test sequence s1 at the variation point, since s1 is, up to this point, common for all feature variants. Test sequence s2 ends with ‘NavToAdrbook’ because firstly, the view state has no outgoing transitions and secondly, it is a breaking-function state. Test sequences s4 and s7 end at the joining point, since they describe market variant specific behavior up to the joining point and from the joining point, the described behavior is common.

In the SC above, variation points and joining points are used for the market variants which are feature variants. As explained in Section 7.2.2, variation and joining points can also be used to describe variability caused by different functional feature options. For the test generation, it does not matter what type of variability the variation and joining points are used for.

The following pseudo-codes illustrate the introduced test sequence generation algorithm:

```
generateTestSequencesAndSequenceGraph(released functional feature state rfs, coverage criteria cc)
begin
    initialize a test sequence graph gr;
    initialize test sequence list seqList;
    initialize a test sequence curSeq;
    initialize a state startingState;
    set curSeq.status = inProcessing;
    seqList.add(curSeq);
    set startingState = initial(rfs);
    generateRec(curSeq, seqList, startingState, rfs, cc);
end
```

```
generateRec(test sequence curSeq, sequence list seqList, state startingState, released functional feature state rfs, coverage criteria cc)
begin
    //For the first invocation, the starting state is the initial state
    if startingState = initial(rfs) and curSeq = null then gr.add(init);
    //For later invocations, overwrite the starting state with the last state of the current sequence
    if curSeq ≠ null then startingState = curSeq.getLastState();

    //1. If the current starting state has no outgoing transition
    if outgoingTransitions(startingState) = ∅ then
        set curSeq.status = finished;
        create a node n for curSeq and gr.insert(n);
        if seqList contains at least one sequence with status(seq)=inProcessing then
            curSeq = seqList.getFirstSequence(inProcessing);
            generateRec(curSeq, seqList, startingState, rfs, cc);
        else return seqList;
        ...
```

```
   ...
//2. If all outgoing transitions of the starting state are already visited
if for each tr $\in$ outgoingTransitions(startingState): status(tr)=visited then
    set curSeq.status = finished;
    create a node n for curSeq and gr.insert(n);
    if seqList contains at least one sequence with status(seq)=inProcessing then
        curSeq = seqList.getFirstSequence(inProcessing);
        generateRec(curSeq, seqList, startingState, rfs, cc);
    else return seqList;

//3. If there is exactly one outgoing transition tr of the current startingState
//which is unvisited and cond(tr) = true;

    //3.1 If tr points to a variation point
    if target(tr) = vp $\in$ VPoints then
        curSeq.insert(tr);
        set curSeq.status = finished;
        create a node n for curSeq and gr.insert(n);
        create a variation node vn and gr.insert(vn);
        For each $v_i \in (v_0,..., v_k)$ of vp with $1 \geq i \geq k$ do:
            insert an edge $e_i$ after the vn in the test sequence graph;
            startingState = vp;
            generateRec(curSeq, seqList, startingState, rfs, cc);
        if seqList contains at least one sequence with status(seq)=inProcessing then
            curSeq=seqList.getFirstSequence(inProcessing);
            generateRec(curSeq,seqList,startingState,sc,cc);

    //3.2 If tr points to a joining point
    else if target(tr) = jp $\in$ JPoints then
        curSeq.insert(tr);
        set curSeq.status = finished;
        create a node n for curSeq and gr.insert(n);
        if there is still no junction node created for jp in the test sequence graph then
            create a junction node jn and gr.insert(jn);
        if seqList contains $\geq$ one sequence with status(seq)=inProcessing then
            curSeq=seqList.getFirstSequence(inProcessing);
            generateRec(curSeq,seqList,startingState,sc,cc);

    //3.3 If tr points to a breaking-function view state
    else if BreakingFunction(target(tr)) = true then
        curBreakingState = target(tr);
        curSeq.insert(tr); curSeq.insert(target(tr));
        set curSeq.status = finished;
        create a node n for curSeq and gr.insert(n);
        if seqList contains $\geq$ one sequence with status(seq)=inProcessing then
            curSeq=seqList.getFirstSequence(inProcessing);
            generateRec(curSeq,seqList,startingState,sc,cc);
        generateRec(null,seqList,curBreakingState,sc,cc);
         ...
    ...
    //3.4 If tr points to a broken-function state
    else if BrokenFunction(target(tr)) = true then
        curBrokenState = target(tr);
        curSeq.insert(tr);
        set curSeq.status = finished;
        create a node n for curSeq and gr.insert(n);
        if seqList contains $\geq$ one sequence with status(seq)=inProcessing then
            curSeq=seqList.getFirstSequence(inProcessing);
            generateRec(curSeq,seqList,startingState,sc,cc);
        generateRec(null,seqList,curBrokenState,sc,cc);

    else //3.5 If tr points to a normal state
        if startingState $\neq$ init(rfs) then
            curSeq.insert(tr);
        curSeq.insert(target(tr));
        if cc is fulfilled for rfs then return seqList;
        else generateRec(curSeq,seqList,startingState,sc,cc);

  //4. If there are more than one unvisited outgoing transitions of startingState
  let T = the set of transitions in outgoingTransitions(startingState) with
   status(tr) = unvisited and cond(tr) = true; let n = T.size();
  if n > 1 then
    copy curSeq for n-1 times and let TST be the set of copied sequences;
    for the k-th transition tr $\in$ T and k-th seq $\in$ TST with $0 \leq k \leq$ TST.size()-1:
        seq.insert(tr);
        if target(tr) is not a variation node then
            seq.insert(target(tr));
        seqList.add(seq); set seq.status = inProcessing;
    for the last tr $\in$ T
        curSeq.insert(tr);
        if target(tr) is not a variation node then
             curTest.insert(target(tr));
    if cc is fulfilled then return seqList;
    else generateRec(curSeq,seqList,startingState,sc,cc);
end
```

Identically to the algorithm introduced in Section 6.2, the algorithm above distinguishes between the four cases; i.e. in which the current state to be processed by the generation algorithm has no transition, only visited transitions, exactly one transition, or more than one unvisited transitions. The specific feature is that the algorithm takes variation points, joining points, breaking-function and broken-function states into account and cuts the current sequence before or after them. Simultaneously, a test sequence graph is built. The pseudo-code above was not engaged in operations creating the test sequence graph. The method gr.inser(n) is assumed to insert the node n into the correct position of the graph and bind n with an edge if it is required.

8.1.2. Constructing tests from test sequences

So far, we have produced the following results:

- A list of global transition entries
- For each product to be tested, an inter-function graph
- For each functional feature to be tested:
  - a test sequence graph
  - a set of test sequences

The introduced test sequence generation was based on the transition coverage. In this section it is explained how tests can be constructed from these results for a given product to be tested. The constructed tests should also together fulfill the transition coverage. The demonstration is based on the ongoing example; tests shall be created for the functional feature navigation for Products 2 and 3 shown in Figure 8.1.

Step 3 Restore dependencies

In step 1, dependencies on global transitions and other relevant functional feature states were released for a functional feature state. They were preserved with the help of global transition entries and an inter-function graph. Test sequences were generated without them. This means that the generated test sequences still do not fulfill the chosen coverage criteria, in our example the transition coverage. Now, in order to construct tests covering all transitions, these dependencies must be considered by the test construction.

Step 3.1 Restore dependencies on global transitions
For each global transition entry globalTr_entry = [event, target, cons] $\in$ GobalTREntries, create a test sequence testSeq = (Cons, Steps) with Cons = (cons), Steps = $\{step_0, step_1\}$, $step_0$ = event and $step_1$ = target. For each created test sequence, create a node and insert it randomly or according to some rules into the test sequence graph.

As explained in Step 1, global transitions are triggerable from any state of the system. It is neither logical nor possible to test every global transition from every system state. The implementation of a global transition is correct if it can be triggered from one arbitrary state. Therefore, we have found the trade-off between testing effort and test quality, i.e. we test each global transition from one system state only. To do this, a created test sequence for a global transition entry needs to be added into the test sequence graph only once. The insertion position is not important and can be individually defined. We propose to insert it after the first nodes under the initial node in order to avoid unnecessarily long tests and execution times.

Created test sequences for the global transition entries shown in Section 8.1.1 are:

g1: (HardKey_Navigation) $\rightarrow$ [NavWizard], Cons = {[Navi is installed]}
g2: (HardKey_Radio) $\rightarrow$ [RadioSenderList]
g3: (HardKey_Telephone) $\rightarrow$ [TelephoneMain], Cons = {[Telephone is installed]}

Test sequences g1, g2 and g3 should be inserted into the test sequence graph shown in Figure 8.10. As explained above, in order to avoid unnecessarily long tests and hence unnecessary execution time, we randomly distribute the three test sequences after the nodes s1 and s2, which are the first nodes after the initial node.

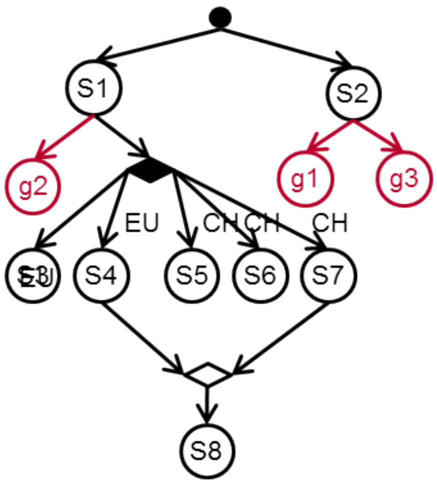


Figure 8.11.: Test sequence graph with nodes presenting global transitions

In this way, global transitions are covered by constructed tests.

Step 3.2 For each relevant functional feature, restore dependencies on other functional feature states
Assume that ff ∈ FFeatures is the functional feature to be tested. For each inter-function transition entry intTransition_entry = [event, source, target, cons] ∈ GetDependentTransitionEntries(ff):
1. create a test sequence testSeq = (Cons, Steps) with Cons = {cons}, Steps = (step$_0$, step$_1$) where step$_0$ = event and step$_1$ = target
2. obtain the set of nodes whose associated sequences end with state source from the test sequence graph: GetNodesEndingWith(source)
3. for each node n ∈ GetNodesEndingWith(source), create a node for testSeq and insert it after n

Dependencies between different functional feature states are realized via inter-function transitions. In Step 1, inter-function transitions were ignored for the test generation. Similarly to global transitions, inter-functional transitions have be considered by the test construction in order to fulfill the transition coverage. This step extends the test sequence graph with nodes representing inter-function transitions.

The inter-function graph for Products 2 and 3 is presented in Figure 8.5. For the functional feature navigation, there is one inter-function transition entry preserved in the graph, which breaks out from the functional feature navigation and leads the HMI to another functional feature state: NavToAdrbook$\xrightarrow{\text{confirm}}$ TelContact. The created test sequence for it is:

i1: (confirm) → [TelContact]

The inter-function transition entry in the graph TelContact $\xrightarrow{\text{startGuidance}}$ NavCalculation, which leads the HMI from the functional feature telephone to navigation, will be considered by the test construction for telephone, but not here.

In the inter-function graph, node s2 is the only node which ends with the source state of the inter-function transition entry 'NavToAdrbook'. Therefore, a node associated with the create test sequence i1 is created and inserted after node s2. Figure 8.12 shows the result of this step.

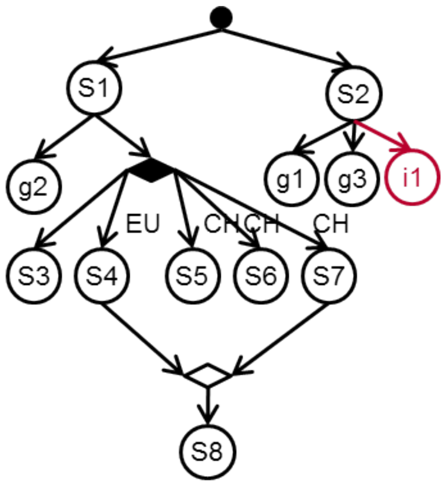


Figure 8.12.: Test sequence graph inserted with an inter-feature transition

If this step is performed for each functional feature to be tested, all inter-function transitions will be covered.

Step 4 Constructing tests

In this step, tests for a particular functional feature to be tested are constructed based on its test sequence graph.

So far, the test sequences contained in the available test sequence graphs together fulfill the transition coverage. In order to create tests covering all transitions, each test sequence preserved in the sequence graph must be contained in at least one of the tests to be created. Therefore, each test sequence graph should be traversed based on the node coverage and each resultant path is a test.

Step 4.1 Create paths from a test sequence graph

The algorithm that searches for paths from a test sequence graph is very similar to the algorithm introduced in Step 2 that generates test sequences from a released functional feature state. It traverses the test sequence graph, takes account of variation and joining points, and generates paths based on depth search. In order to generate tests for more than one product, e.g. for Products 2 and 3, the test sequence graph needs to be traversed only once. The searching algorithm is not introduced in detail here.

The following paths are created from the test sequence graph shown in Figure 8.12:

For the European market:
Path 1: s1 - s3
Path 2: s1 - s4 - s8

For the Chinese market:
Path 3: s1 - s5
Path 4: s1 - s6
Path 5: s1 - s7 - s8

Common for both markets:
Path 6: s1 - g2
Path 7: s2 - g1
Path 8: s2 - g3
Path 9: s2 - i1

If global transitions and inter-function transitions are not required for testing, tests should be generated from the test sequence graph shown in Figure 8.10. In this case, there is only one common test for both markets:

Path 10: s2

Tests for Product 2 are composed of the paths for the Chinese market and the common paths. Tests for Product 3 are composed of the paths for the European market and the common paths.

Step 4.2 Instantiate created paths into tests

Instantiated tests from the above paths are as follows:

Tests for the European market:

Test 1:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput]

Test 2:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap] → (SK_Route) → [NavWizard]

Tests for the Chinese market:

Test 3:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput]

Test 4:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (province) →

[NavProvinceList] → (chooseProvince) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput]

Test 5:
[Main] → (nav) → [NavWizard] → (address) → [NavAdrInput] → (province) → [NavProvinceList] → (chooseProvince) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start) → [NavCalculation] → [NavMap] → (SK_Route) → [NavWizard]

Common tests for both market variants:

Test 6:
[Main] → (nav) → [NavWizard] → (address) → (HardKey_Radio) → [RadioSenderList]

Test 7:
[Main] → (nav) → [NavWizard] → (adrbook) → [NavToAdrbook] → (HardKey_Navigation) → [NavWizard]

Test 8:
[Main] → (nav) → [NavWizard] → (adrbook) → [NavToAdrbook] → (HardKey_Telephone) → [TelephoneMain]

Test 9:
[Main] → (nav) → [NavWizard] → (adrbook) → [NavToAdrbook] → (confirm) → [TelContact]

If global transitions and inter-function transitions are not intended for testing, the common test for both market variants is:

Test 10:
[Main] → (nav) → [NavWizard] → (adrbook) → [NavToAdrbook]

8.1.3. Avoiding redundant tests

Until now, it has been introduced how tests can be generated for different variants without the need for the generation process to be repeated. However, generated tests for different products still have a large set of identical tests which verify the common behavior of these products. For instance, Test 6 to Test 10 listed at the end of the last step are contained in the tests generated for both Product 2 and Product 3, and would be executed for both products. Also, the tests generated for the functional feature radio which has no variability in the menu behavior would be executed for all three products.

It can never be assured that the same tests verifying common behavior do not produce new or different errors for different products. Therefore, if sufficient testing resources are available, these tests should be executed for each product. However, in the case of limited testing resources, a trade-off must be made between the testing effort and the quality.

In this step, such identical tests verifying common behavior of different products to be tested will be detected and avoided for the test execution. We assume that tests have been generated for all three products shown in Figure 8.1 and that the products should be tested in the order in which they are graphically displayed.

The functional feature radio is provided by all three products and does not have variability in the menu behavior. Nor does it have any dependencies on other functional features, as shown in Figure 8.5. According to the trade-off principle between the testing effort and quality, it is sufficient to execute generated tests for the functional feature radio only once for Product 1. Furthermore, Products 1 and 2 both provide the functional feature telephone in the same option "BTHandfree". Tests generated for this option should also be executed only once for Product 1 and do not need to be repeated for Product 2. However, since the dependencies between the functional feature telephone and navigation are not available in Product 1, tests verifying these dependencies must be executed for Product 2. Additionally, Test 6 to Test 10 listed at the end of Step 4.2 are generated for both Product 2 and Product 3 which have different market variants. They are variant feature unspecific, i.e. they test the part of the menu behavior which is common for both variants. However, they would be redundantly executed a second time. Similarly, different functional feature options of the same functional feature also can share a large set of common behaviors. Tests generated from common behaviors would also be redundantly executed.

In order to avoid redundant test executions, the following rules must be followed:

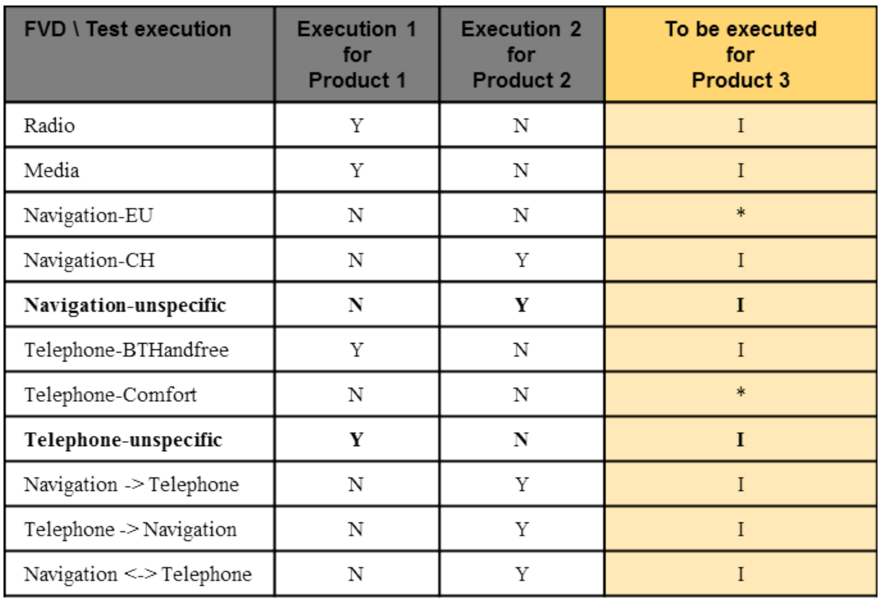
1. Already tested functional features should not be tested again if they have no variability in the menu behavior (e.g. the functional feature radio)
2. If a functional feature has variability in the menu behavior which is caused by variant features, tests for each variant feature should be executed once, and only once (e.g. the functional feature navigation)
3. If a functional feature has variability in the menu behavior which is caused by functional feature options, tests for each functional feature option should be executed once, and only once (e.g. the functional feature telephone)
4. If a functional feature has behavior which is unspecific for any variant features or functional feature options, tests verifying this unspecific behavior should be executed only once
5. Untested dependencies of already tested functional features should be tested (e.g. the dependencies between the functional features telephone and navigation)

Based on the idea in [113], the *FVD-matrix* (F: functional feature, V: variant feature and D: dependencies) is defined to realize the rules above. A FVD-matrix is a matrix which records the already tested entities by performed test executions. Based on the content of the FVD-matrix, it can be calculated which tests should be performed for the next product to be tested. The entities can be:

- a functional feature, if it has no variability in the menu behavior, such as the functional feature radio
- a functional feature option of a functional feature, if this functional feature is provided in more than one option and the current option has no variability in the menu behavior, such as "telephone-BTHandfree" or "telephone-comfort"

- a variant feature of a functional feature, if this functional feature has variability in the menu behavior which is caused by different variant features, such as "navigation-CH" and "navigation-EU"
- a variant feature of a functional feature option, if this functional feature option has variability in the menu behavior which is caused by different variant features
- a functional feature followed by the string "unspecific", if the functional feature has variability, but also unspecific behavior, such as "navigation-unspecific"
- a dependency, such as the dependency of the functional feature navigation on the functional feature telephone

Figure 8.13 shows the FVD-matrix after tests have been executed for Product 1 and Product 2.



FVD \ Test execution	Execution 1 for Product 1	Execution 2 for Product 2	To be executed for Product 3
Radio	Y	N	I
Media	Y	N	I
Navigation-EU	N	N	*
Navigation-CH	N	Y	I
Navigation-unspecific	**N**	**Y**	**I**
Telephone-BTHandfree	Y	N	I
Telephone-Comfort	N	N	*
Telephone-unspecific	**Y**	**N**	**I**
Navigation -> Telephone	N	Y	I
Telephone -> Navigation	N	Y	I
Navigation <-> Telephone	N	Y	I

Figure 8.13.: Entries required for testing Product 3

The entities in the first column can be obtained from test sequence graphs generated in Step 2 and inter-functional graphs generated by Step 1.3. If a test sequence graph contains no variation or junction nodes, then the respective functional feature should be added as an entry into the first column. If the test sequence graph contains variation and junction points used for variant features as shown in Figure 8.12, then for each feature variant, an entry should be added into the first column. Similarly, if a test sequence graph contains variation and junction points used for different functional feature options, then for each functional feature option, an entry should be added. If nested variation and junction nodes are used, first for functional feature options and second for variant features, then an entry should be added for each variant feature of each functional feature option. Dependencies between two functional features can be obtained from the inter-functional graphs (see Step 1.3 in Section 8.1.1). For each edge in the inter-feature graph as shown in Figure 8.5, an entry should be added. A cycle in the graph, which means a two-way dependency can also be added as an entry into the matrix, if for instance path coverage is intended. From the second column, entities already tested by each performed execution are recorded.

Now, if a product $product_n$ is to be tested, entities required to be executed for $product_n$: fvd_n can be calculated based on the FVD-matrix as follows:
$fvd_n = delta_n = fvd_n \setminus fvd_{1...n-1}$ where $fvd_{1...n-1} = fvd_1 \cup ... \cup fvd_{n-1}$.

Required entities are labeled with "*". Entities which can be ignored are labeled with "I". The last column is not part of the FVD-matrix; it merely presents the calculation result.

Without the help of the FVD-matrix, tests for the functional feature radio would be executed twice too often and "telephone-BTHandfree" once to often.

8.1.4. Conclusions

The introduced test generation strategy consists of three steps: first, the generation algorithm traverses the menu behavior SC in its contained released functional feature states only once and generates test sequences which are potentially needed for testing any product provided by the product line. Simultaneously, for each functional feature, a test sequence graph is generated which manages the generated sequences for this functional feature. Second, for one or several concrete products to be tested, the required test sequence graphs must be traversed once, allowing tests to be created for different products all at once. Finally, common tests produced for different products can be identified. In this way, we avoid repeatedly traversing the complete SC for each product to be tested and repeatedly executing the common tests for different products.

8.2. Some other coverage criteria

The introduced test generation strategy was based on the transition coverage. In this section, we discuss the generation based on some other common coverage criteria.

State coverage (Section 2.1.2) requires that generated tests together cover all states in the SC. For state coverage, global transitions and inter-function transitions don't have to be tested, since the view states they are pointing to are usually reachable via other normal transitions. The test generation strategy for state coverage is very similar to the introduced strategy based on transition coverage.

If path coverage (Section 2.1.2) is intended, it is not enough to handle an inter-functional transition in the way introduced in Step 3.2 (8.1.2), since it leads to the loss of possible paths. First, an inter-function transition leads to paths which do not stop at the end of this transition, but continue beyond it into the other functional feature state. Furthermore, round-trip inter-function transitions such as the ones between the functional features navigation and telephone (shown in Figure 8.5) must especially be considered, since together they lead to a path which first breaks out from the current functional feature, then visits another functional feature and finally returns to the current functional feature.

Therefore, if path coverage is intended, functional feature states which have dependencies on each other must first be considered as one released function feature state by the test generation, as shown in Figure 8.14. Tests should be generated from such a joined released functional feature state in order to find all possible paths. Second, these functional feature states having dependencies on each other must also be separately considered by the test generation as introduced so far, since there can be products to be tested which do not provide both functional feature states, such as Product 1 in Figure 8.1.

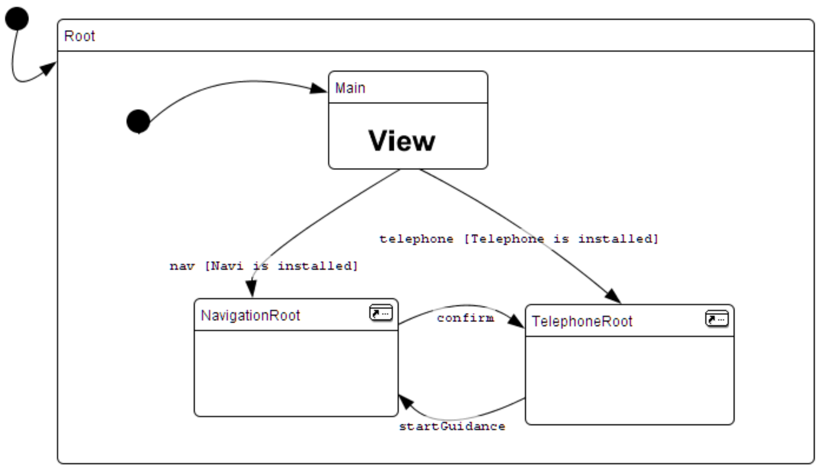


Figure 8.14.: Joined released functional feature state for the functional features navigation and telephone

8.3. Conclusions

In this chapter, it was introduced how menu behavior tests can be generated from a menu behavior SC describing variability for different products of a PL.

Different HMI products share a large set of common functional features and menu behaviors. If the test generation must be repeatedly performed for each product to be tested, the common menu behaviors of shared functional features must be requested as often as the number of products to be tested. Due to the high degree of commonality and the large number of products in practice, the challenge that we present to ourselves, i.e. to avoid redundant test executions, is a sizeable one. A test generation strategy has been introduced which consists of the following two steps: first, the generation algorithm traverses the menu behavior SC only once in its contained released functional feature states and generates test sequences which are potentially needed for testing any product provided by a product line. Simultaneously, for each functional feature, a test sequence graph is generated which manages the generated sequences for this functional feature. Second, for one or several concrete products to be tested, the required test sequence graphs must be traversed only once for tests to be created for different products all at one time. In this way, for potential products to be tested, the complete SC, which is usually gigantic in practice, must be traversed only once. The generated test sequence graphs, which are much smaller than a SC, also must be searched only once.

The introduced test generation strategy was based on transition coverage. We also briefly discussed test generation with variability based on state coverage and path coverage. It was shown that the test generation strategy must be adapted for other coverage criteria. Although we could not present all possible coverage criteria in this work, we hope to provide inspiration to both research and industry via the proposed approach.

Part IV.

Evaluation and Conclusions

This is the final part of the thesis. In this section, the variability approaches introduced in III will be evaluated. We will also give a summary of the focuses of this thesis and the proposed solutions. Even though model-based testing of infotainment system HMIs has recently become a hot topic in the automotive industry and in research, there are still a large number of open research questions to be answered. Finally, we will introduce future work in the domain of model-based HMI testing from the perspectives of the industry and research.

Chapter 9.

Evaluation

The main focal points of this thesis are firstly modeling the menu behavior with variability (Section 7.2) and secondly generating menu behavior tests without redundancies for potential products which are to be tested (Chapter 8).

The purpose of this chapter is to evaluate these approaches. The evaluation will be done from the efficiency aspect by estimating the efficiency improvement, i.e. the time saved by using our approaches. For this estimation, we use the data which is obtained from our proof-of-concept implementation and an HMI development model (Section 2.2.3) from real life. Both data sources will be introduced in Section 9.1. The evaluation consists of mathematical analyses which prove the plausibility of the proposed approaches, general discussions about factors influencing efficiency, and also concrete data demonstrating the efficiency improvements.

9.1. Data source for the evaluation

In this section, we introduce our proof-of-concept implementation and an HMI development model from real life. We perform measurements based on the proof-of-concept implementation and attend to several numbers from the practical HMI development model. Based on the comparison of these data, we afterwards estimate how the efficiency of testing a real-life HMI product line can be improved by using our approaches.

9.1.1. Proof-of-concept implementation

In order to prove the plausibility of the concept, we have implemented a strongly simplified product line, a menu behavior SC with variability, and a test generator which generates tests from this SC without redundancies. These implementations have already been introduced as the ongoing example in earlier chapters. We give a summary of them at this point. The figures shown again here do not contain any changes.

Proof-of-concept product line

Figure 9.1 shows the extended feature model describing our proof-of-concept product line.

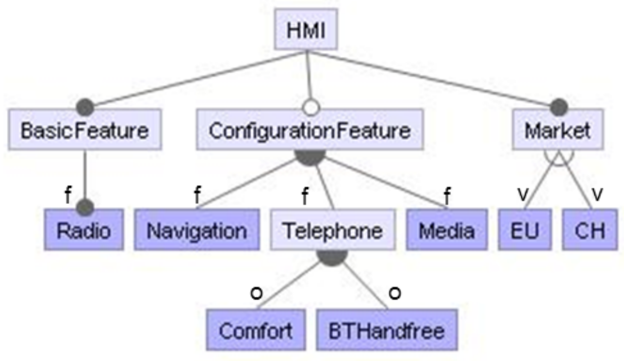
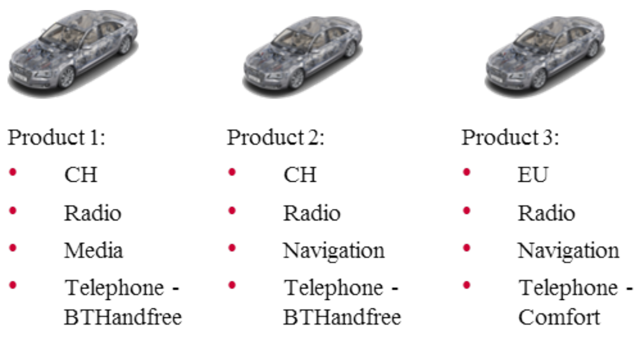


Figure 9.1.: EFM of the proof-of-concept product line

The product line shown above provides four functional features: radio which is provided in each product as a basic feature, and navigation, telephone and media which can be optionally embedded in a product. Functional features are labeled with “f” in the EFM. The functional feature telephone can be provided in two possible functional feature options: “comfort” and “BTHandfree” which are labeled with “o” in the EFM. The product line also provides two possible market variants which are feature variants: the European market and the Chinese market which are labeled with “v” in the EFM. Three products are provided by this product line, as shown in Figure 9.2.


Product 1:
- CH
- Radio
- Media
- Telephone - BTHandfree

Product 2:
- CH
- Radio
- Navigation
- Telephone - BTHandfree

Product 3:
- EU
- Radio
- Navigation
- Telephone - Comfort

Figure 9.2.: Three products provided by the proof-of-concept product line

Proof-of-concept menu behavior SC

The proof-of-concept menu behavior SC describes a very simplified menu behavior with variability. The root state of the SC is represented in Figure 9.3.

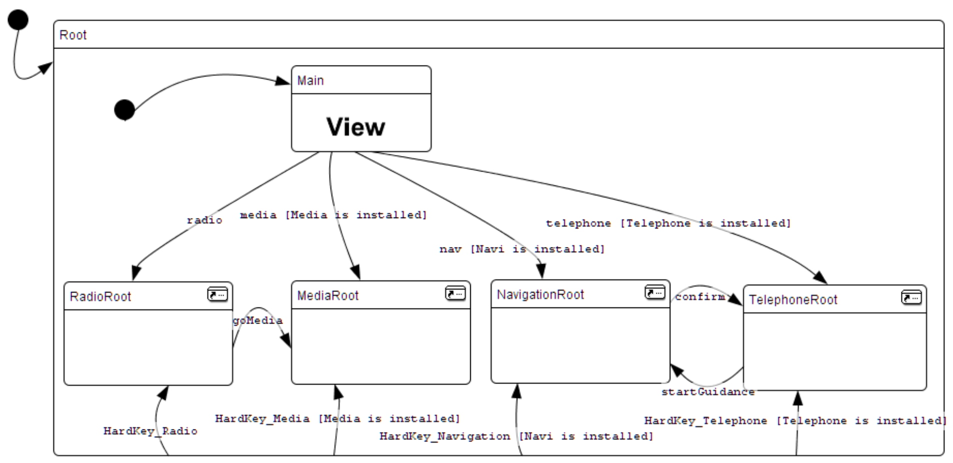

Root
Main
View
radio
media [Media is installed]
nav [Navi is installed]
telephone [Telephone is installed]
RadioRoot
MediaRoot
NavigationRoot
TelephoneRoot
goMedia
confirm
startGuidance
HardKey_Radio
HardKey_Media [Media is installed]
HardKey_Navigation [Navi is installed]
HardKey_Telephone [Telephone is installed]

Figure 9.3.: Root state of the proof-of-concept menu behavior SC with variability

The root state shown above contains four functional feature states for the four functional features defined in the EFM. The content of the functional feature state ‘NavigationRoot’ is represented in Figure 9.4. Other functional feature states are not modeled in full, since the test generation should be performed only for the functional feature navigation.

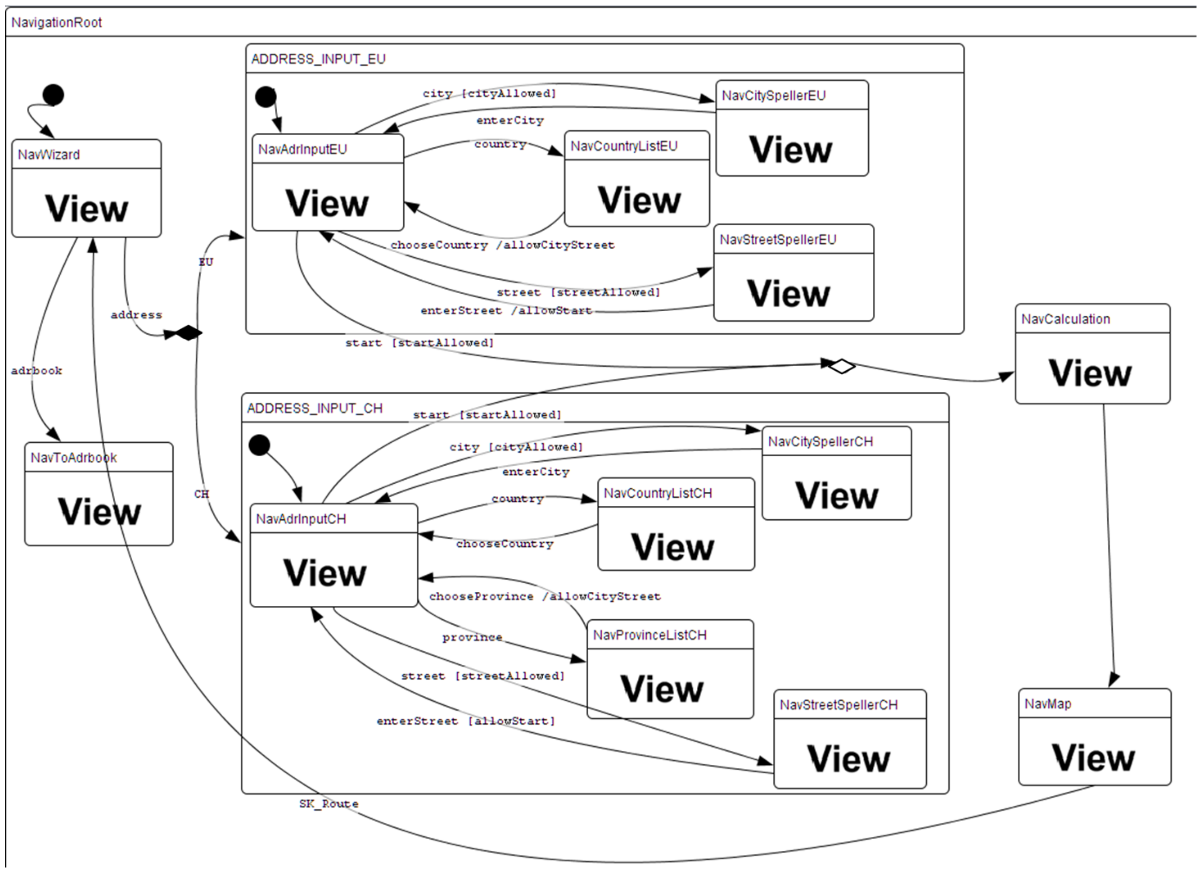


Figure 9.4.: Functional feature state of the functional feature navigation

Proof-of-concept test generator

The generation algorithm based on which the test generator is implemented has already been introduced in Section 8.1. It is based on the depth-first search and transition coverage. The generator allows each cycle only once. In Section 6.2, it has been explained that hierarchies of SCs are resolved in advance and the test generator receives SCs with a flat structure as input.

Step 1 and step 2, introduced in Subsection 8.1.1, are not implemented as separate steps in the proof-of-concept generator. The separation is only to facilitate the introduction of the algorithm.

The following test sequences are created for the functional feature navigation, if global transitions and inter-function transitions are not intended to be tested:

s1:
[Main] → (nav) → [NavWizard] → (address)

s2:
[Main] → (nav) → [NavWizard] → (adrbook) → [NavToAdrbook]

s3:
[NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput]

s4:
[NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start)

s5:
[NavAdrInput] → (country) → [NavCountryList] → (chooseCountry) → [NavAdrInput]

s6:
[NavAdrInput] → (province) → [NavProvinceList] → (chooseProvince) → [NavAdrInput] → (city) → [NavCitySpeller] → (enterCity) → [NavAdrInput]

s7:
[NavAdrInput] → (province) → [NavProvinceList] → (chooseProvince) → [NavAdrInput] → (street) → [NavStreetSpeller] → (enterStreet) → [NavAdrInput] → (start)

s8:
[NavCalculation] → [NavMap] → (SK_Route) → [NavWizard]

The respective test sequence graph is shown in Figure 9.5.

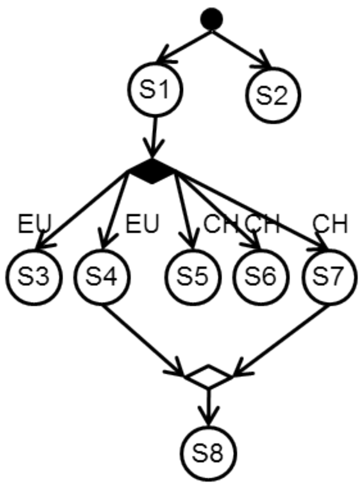


Figure 9.5.: Test sequence graph of functional feature navigation

Path s1 - s3, path s1 - s4 - s8 and path s2 comprise the tests for the European market. Path s1 - s5, path s1 - s6 and path s1 - s7 - s8 comprise the tests for the Chinese market.

9.1.2. An HMI development model from real life

The efficiency estimation which should be done in order the show the plausibility of our concepts requires not only concrete data which can be measured based on the proof-of-concept implementation but also data from a real-life HMI project. For this, we use an HMI development model (Section 2.2.3) from real life which has been developed for the car model A3. In this development model, the variation is described with conditions which cannot be distinguished from other conditions used for conventional goals such as verifying the validity of user inputs or readiness of an underlying application. Also, there are no feature models or similar mechanisms to define the product line, provided products and the relations between them. These two facts however do not affect the correctness of the generated HMI software. The code generator generates the same codes for all provided products and the underlying applications are responsible for assigning the values of the conditions during runtime. For instance, via the value assignments, the underlying applications “tell” the HMI for which market variant it is intended and which functions are available in the current product. The HMI then behaves according to the fulfillment of these conditions just as according to other conventional conditions.

The menu behavior SC in such a development model is unsuitable for test generation. First, as introduced in Section 5.2, the SC does not contain sufficient data to generate valid tests. Second, due to the missing concept of specifying variability in the menu behavior SC, the test generator cannot identify which menu behavior is valid for which products. Therefore the test generator would generate a large number of invalid tests.

Nevertheless, we can use the menu behavior SC of this development model for our estimation; we do not have to remodel it according to our modeling approach, since our goal is merely to obtain data from it.

Before we introduce the concrete obtained data, we will first introduce this development model. To do this, we have manually created an EFM describing the product line which is implicitly described in this development model. Figure 9.6 shows this EFM. In order to avoid unnecessary complexity and due to confidentiality requirements, a few of the functional feature options provided are hidden or simplified.

As shown in the EFM in Figure 9.6, the practical product line provides many more market variants than the proof-of-concept product line. Additionally, it provides different system variants which are also variant features. A "standard" system has a smaller display, smaller screens and a lower resolution whereas a "high" system has a bigger display, bigger screens and a high resolution. In this EFM, we decided not to list the four provided car models which are also variant features due to space limitation and to avoid an unnecessary increase in complexity for the estimation. Functional features under "SerialFunctions" are equipped as standard in all cars. Functional features under "ConfigurableFunctions" can be individually chosen. As the EFM shows, in practice most of the functional features can be provided in different options. There are complex relationships between functional features, functional feature options and variant features; e.g. a "high" system implies "NavHigh" and "DVDTwoSD". We will not discuss these relationships further at this point.

The products planned for this product line are presented in Figure 9.7. In the table, "x" stands for "standard". An entry with "x" is always integrated except when an alternative is available which is labeled with "o". Label "o" stands for "optional". Entries with "o" are not planned as standard but can be integrated on the demand of the customers. Entries with "-" are "not combinable" for the current product. Altogether, 86 products are defined for this product line.

The third row in Figure 9.7 represents 6 products provided by the real-life product line. We demonstrate these 6 products:

Product 1: Basic amplifier and no telephone (standard)
Product 2: Basic amplifier and comfort
Product 3: Basic amplifier and BT-handfree
Product 4: Premium amplifier and no telephone
Product 5: Premium amplifier and comfort
Product 6: Premium amplifier and BT-handfree

In practice, it is unusual to test all 86 products in full. First, it is quite difficult to provide 86 cars or test benches, and it requires extreme testing efforts and a high demand on human and time resources. Moreover, according to the *Pareto-Zipf–type laws*, which also exist in software testing [31], it is not necessary to test each of the products in full. Fenton has stated in [33] that 60% of all errors are contained in only 20% of the modules of their telecommunication switching system. For instance, if Product 2 and Product 6 listed above are tested, all "amplifier" and "telephone" options are covered and it is quite probable that most of the errors will be detected. Testing the other 4 products may detect new errors; however, the benefit is quite low in proportion to the testing effort. Currently, in the Audi testing laboratory, 14 test benches are available of which 6 are for "standard" systems and 8 are for "high" systems. Together they cover all the functional features, functional feature options and variant features provided, as well as their most important combinations.

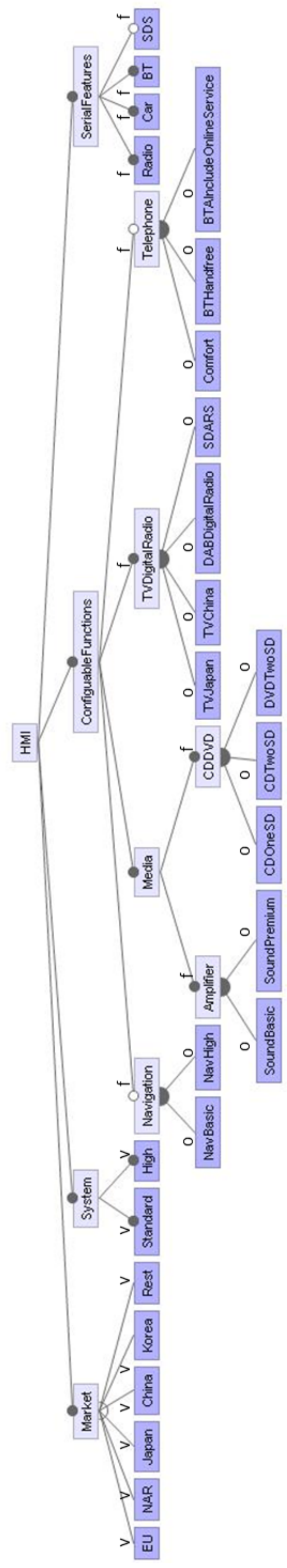
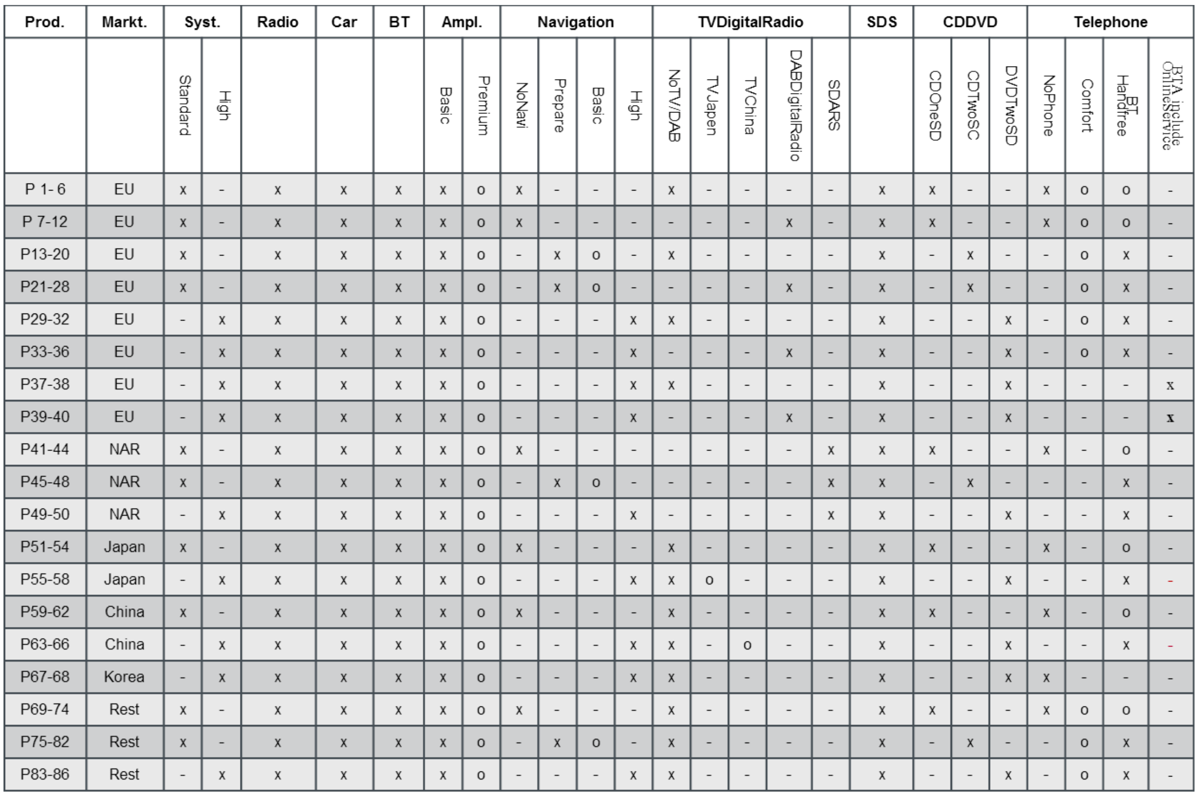


Figure 9.6.: EFM describing the practical product line

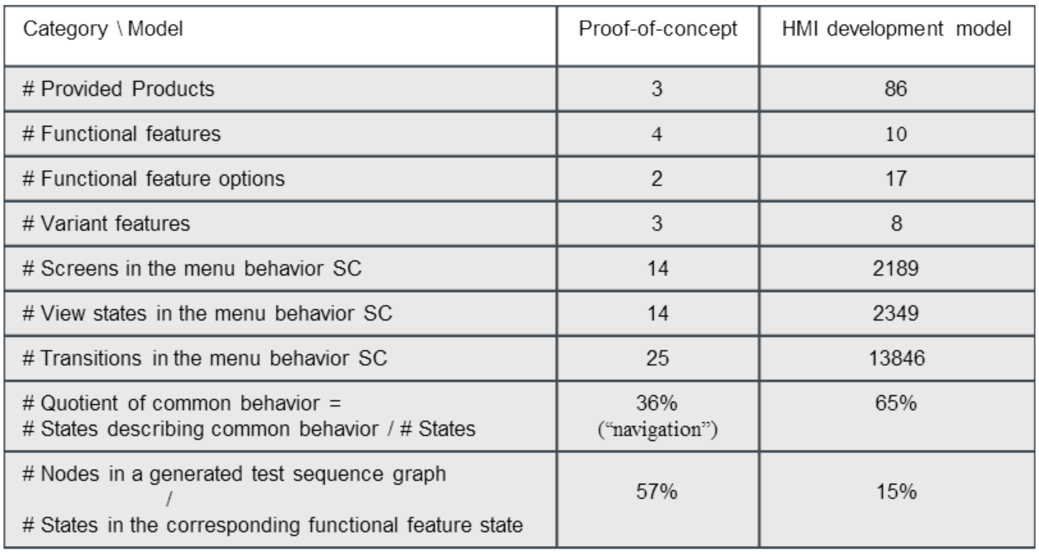


Prod.	Markt.	Syst.		Radio	Car	BT	Ampl.		Navigation				TVDigitalRadio					SDS	CDDVD			Telephone			
		Standard	High				Basic	Premium	NoNavi	Prepare	Basic	High	NoTV/DAB	TVJapen	TVChina	DABDigitalRadio	SDARS		CDOneSD	CDTwoSC	DVDTwoSD	NoPhone	Comfort	BT Handfree	BTA include OnlineService
P 1- 6	EU	x	-	x	x	x	x	o	x	-	-	-	x	-	-	-	-	x	x	-	-	x	o	o	-
P 7-12	EU	x	-	x	x	x	x	o	x	-	-	-	-	-	-	x	-	x	x	-	-	x	o	o	-
P13-20	EU	x	-	x	x	x	x	o	-	x	o	-	x	-	-	-	-	x	-	x	-	-	o	x	-
P21-28	EU	x	-	x	x	x	x	o	-	x	o	-	-	-	-	x	-	x	-	x	-	-	o	x	-
P29-32	EU	-	x	x	x	x	x	o	-	-	-	x	x	-	-	-	-	x	-	-	x	-	o	x	-
P33-36	EU	-	x	x	x	x	x	o	-	-	-	x	-	-	-	x	-	x	-	-	x	-	o	x	-
P37-38	EU	-	x	x	x	x	x	o	-	-	-	x	x	-	-	-	-	x	-	-	x	-	-	-	x
P39-40	EU	-	x	x	x	x	x	o	-	-	-	x	-	-	-	x	-	x	-	-	x	-	-	-	x
P41-44	NAR	x	-	x	x	x	x	o	x	-	-	-	-	-	-	-	x	x	x	-	-	x	-	o	-
P45-48	NAR	x	-	x	x	x	x	o	-	x	o	-	-	-	-	-	x	x	-	x	-	-	-	x	-
P49-50	NAR	-	x	x	x	x	x	o	-	-	-	x	-	-	-	-	x	x	-	-	x	-	-	x	-
P51-54	Japan	x	-	x	x	x	x	o	x	-	-	-	x	-	-	-	-	x	x	-	-	x	-	o	-
P55-58	Japan	-	x	x	x	x	x	o	-	-	-	x	x	o	-	-	-	x	-	-	x	-	-	x	-
P59-62	China	x	-	x	x	x	x	o	x	-	-	-	x	-	-	-	-	x	x	-	-	x	-	o	-
P63-66	China	-	x	x	x	x	x	o	-	-	-	x	x	-	o	-	-	x	-	-	x	-	-	x	-
P67-68	Korea	-	x	x	x	x	x	o	-	-	-	x	x	-	-	-	-	x	-	-	x	x	-	-	-
P69-74	Rest	x	-	x	x	x	x	o	x	-	-	-	x	-	-	-	-	x	x	-	-	x	o	o	-
P75-82	Rest	x	-	x	x	x	x	o	-	x	o	-	x	-	-	-	-	x	-	x	-	-	o	x	-
P83-86	Rest	-	x	x	x	x	x	o	-	-	-	x	x	-	-	-	-	x	-	-	x	-	o	x	-

Figure 9.7.: Products of the practical HMI development model

In order to do a realistic evaluation, our estimation later will be based on the assumption that 14 of the 86 products are intended to be tested and that these 14 products cover all functional features, functional feature options and variant features. Later discussion will show that the greater the number of products tested, the more efficient our approaches.

The table shown in Figure 9.8 compares the proof-of-concept menu behavior SC with the menu behavior SC from the practical development model:



Category \ Model	Proof-of-concept	HMI development model
# Provided Products	3	86
# Functional features	4	10
# Functional feature options	2	17
# Variant features	3	8
# Screens in the menu behavior SC	14	2189
# View states in the menu behavior SC	14	2349
# Transitions in the menu behavior SC	25	13846
# Quotient of common behavior = # States describing common behavior / # States	36% ("navigation")	65%
# Nodes in a generated test sequence graph / # States in the corresponding functional feature state	57%	15%

Figure 9.8.: Proof-of-concept vs. the practical HMI development model

The "quotient of common behavior" entry represents the percentage of the total number of states in the SC that are states describing common behavior. It is used to describe the commonality of the products provided by a PL.

In the proof-of concept implementation, the flattened SC describing the menu behavior of the functional feature navigation which was shown in Figure 8.8 contains 14 states altogether. Only 5 of them describe the common behavior of the EU and CH markets. From this we can deduce 36% commonality.

In comparison, the commonality of the menu behavior in the practical model is much higher. A manual evaluation of the HMI development model has shown that about 65% of the menu behavior is common for the provided products. This percentage has been calculated as follows: the first component of the commonality comes from the functional features radio, car, BT and SDS which are standard functional features and have hardly any variability in their menu behaviors. They make up about 9% of the complete menu behavior of the HMI. The second component of the commonality comes from the functional features amplifier and CDDVD. As shown in Figure 9.1, each of these is provided in one of their available options in each provided product. The differences between the menu behaviors of the available options are minimal. Thus, the functional features amplifier and CDDVD make up about 8% of the commonality. The third component comes from the functional features navigation and telephone which have the most complex menu behaviors and require a large proportion of the states to describe them. About 50% of the states in the SC describe the menu behavior of the functional feature navigation. Although navigation is not provided in all products, it is provided in 62 of the 86 products, i.e. about 70% of available products. The menu behaviors of the options “NavBasic” and “NavHigh” have a large set of commonalities: about 95%. Therefore, we calculate that 50% * 70% * 95% = 34% of the menu behavior of the functional feature navigation is the common behavior. Similarly to navigation, the functional feature telephone requires about 25% of the states to describe its menu behavior and is provided in 72 of the 86 products (84%). The menu behaviors of different functional feature options contain about 65% common behaviors. Thus, telephone adds 25% * 84% * 65% = 14% common behavior. Altogether, about 9% + 8% + 34% + 14% = 65% of the menu behaviors are common in the development model.

In the proof-of-concept implementation, the sequence graph generated for navigation contains 8 nodes (Figure 9.5), i.e. about 57% of the states from which the sequence graph is generated. This percentage is actually unrealistic. In the practical menu behavior SC, the number of the nodes in a test sequence graph is only about 15% of the number of the states from which the sequence graph is generated. This means that a generated test sequence graph is much smaller than the functional feature state from which the graph is generated.

9.2. Evaluation

In this section, the approaches of modeling menu behavior with variability (introduced in Section 7.2) and generating tests without redundancies (introduced in Chapter 8) will be evaluated.

The evaluation consists of the following three components: first, in Section 9.2.2, it will be mathematically calculated that the proposed approaches always achieve an efficiency improvement for a chosen practical HMI project. Second, in Section 9.2.3, there will be a general discussion about which factors the efficiency improvement is dependent upon. In addition, worst cases will be identified in which the approaches are not profitable. Finally, in Section 9.2.4, an estimation with concrete data will be made in order to demonstrate how much time would be saved for a project in practice by using our approaches.

First, the design of the evaluation will be introduced.

9.2.1. Design of the evaluation

We compare the efficiency for the following methods:

Method 1: A menu behavior SC is available which describes variability based on the modeling approach introduced in Section 7.2. A test generation method is available which generates tests from this SC for different products of the PL and avoids redundancies as introduced in Chapter 8.

Method 2: A menu behavior SC is available which describes variability based on the modeling approach introduced in Section 7.2. The test generation used does not avoid redundancies. This means that for each product to be tested, it traverses all relevant parts of the SC.

Method 3: A menu behavior SC is available which has no clear concept to describe variability, similar to the one in the practical HMI development model. Consequently, the described variability is untraceable for the test generation. The test generation nevertheless generates tests from this SC and cannot avoid redundancies.

The primary goal is to show that the test generation time of Method 1 is better than the test generation time of Method 2 based on a mathematical analysis. Thus the efficiency of the proposed generation method which avoids redundant test generations is evaluated. Second, if it is possible to show that the generation time of Method 2 is better than that of Method 3, then the efficiency of the proposed method modeling variability is evaluated. We will first show these statements based on mathematical analyses. Then we will discuss generally on which factors the efficiency improvements of our proposed approaches are dependent. Finally, we will give an estimate of the saved generation time for these three methods using concrete data.

For the evaluation, we assume that a systematic test coverage is set as the testing goal by choosing several products for testing which cover all functional features, variant features and functional feature options. We assume that n is the number of chosen products to be tested and n > 1, since it is unrealistic that only one product of a product line is to be tested.

For the test generation we use flattened SCs.

In order to compare the test generation times of the three methods, we present the required generation time of each method as an equation.

For Method 1, the generation algorithm needs to traverse the menu behavior SC only once and prepares all potentially needed test sequences and test sequence graphs. Due to the assumption that the chosen products to be tested cover all functional features, all generated test sequence graphs have to be traversed once. The equation describing the generation time can be shown as:

$$T_{generation\_c1} = T_{traverseCompleteSC} + T_{traverseAllGraphs}$$

According to the test generation method introduced in 8.1, if more products are to be tested, $T_{generation\_c1}$ remains the same. If a subset of the products are to be tested which do not cover all functional features, then not all of the generated test sequence graphs have to be traversed. The generation time is smaller than $T_{generation\_c1}$.

For Method 2, a menu behavior SC is available which describes variability based on the proposed approach. The test generation is a normal test generation which is not intended to avoid redundancies. We assume that the test generation algorithm takes the elements describing variability into account and only traverses the SC in parts which are relevant for the current product. For each product, the test generation algorithm has to be performed once. Therefore, the test generation time can be calculated as:

$$T_{generation\_c2} = T_{traversePartSC\_product_1} + \dots + T_{traversePartSC\_product_n}$$

where n is the number of products to be tested.

For Method 3, the complete SC must be traversed once for each product, since the SC does not contain elements indicating common or product specific behaviors for the test generator.

As explained previously, since the test generation algorithm produces invalid tests, manual adaption of generated tests is required. Therefore, the test generation time of Method 3 is composed of traversing the complete menu behavior SC as often as the number of products which are to be tested plus manual correction of the generated tests. We do not yet have any experience of how time consuming the manual work can be. However, the effort would become unmanageable as the number of products which are to be tested increases.

$T_{generation\_c3}$ = n * $T_{traverseCompleteSC}$ + n * T_{manual}

where n is the number of products to be tested.

9.2.2. Evaluation based on mathematical plausibility analysis

In this section, we mathematically show that based on the real-life HMI development model and the assumption that the chosen products to be tested cover all functional features, the test generation time of Method 1 is better than the test generation time of Method 2, and the test generation time of Method 2 is better than that of Method 3.

Statement 1: $T_{generation\_c2} > T_{generation\_c1}$
Proof:
The time T of depth search is linear [URLc]. Consequently:
1. *Since the complete menu behavior is composed of common and specific behaviors:*
$T_{traverseCompleteSC} = T_{traverse\_common} + T_{traverse\_specific}$
2. *Since the common behavior amounts to 65% of the total behavior (Table 9.8):*
$T_{traverse\_common} = T_{traverseCompleteSC}$ *65%
3. *Since the nodes in generated test sequence graphs are about 15% of the states in the SC:*
$T_{generation\_c1} = T_{traverseCompleteSC} + T_{traverseAllGraphs}$
$= T_{traverseCompleteSC}$ + 15% * $T_{traverseCompleteSC}$

For Method 2, the common behaviors are visited n times and the rest, which describe product specific behaviors, are visited at least once completely. Consequently:
$T_{traversePartSC\_product_1}+\ldots+T_{traversePartSC\_product_n} \geq$ n * $T_{traverse\_common} + T_{traverse\_specific}$
where n is the number of products to be tested and n > 1

Consequently:

$$\begin{aligned}
T_{generation\_c2} &= T_{traversePartSC\_product_1} + \ldots + T_{traversePartSC\_product_n}\\
&\geq n * T_{traverse\_common} + T_{traverse\_specific}\\
&= (T_{traverse\_common} + T_{traverse\_specific}) + (n\text{-}1) * T_{traverse\_common}\\
&= T_{traverseCompleteSC} + (n\text{-}1) * T_{traverse\_common}\\
&> T_{traverseCompleteSC} + T_{traverse\_common}\\
&= T_{traverseCompleteSC} + 65\% * T_{traverseCompleteSC}\\
&>> T_{traverseCompleteSC} + 15\% * T_{traverseCompleteSC}\\
&= T_{traverseCompleteSC} + T_{traverseAllGraphs}\\
&= T_{generation\_c1}
\end{aligned}$$

Therefore: $T_{generation\_c2} > T_{generation\_c1}$

$T_{generation\_c2} > T_{generation\_c1}$ shows that for the HMI development project from real life and based on the assumption that the chosen products to be tested cover all provided functional features, functional feature options and variant features, the proposed generation method which avoids redundant test generations saves generation time and hence achieves an improvement in efficiency.

Statement 2: $T_{generation\_c3} > T_{generation\_c2}$

Proof:

$$\begin{aligned} T_{generation\_c3} &= \text{n} * T_{traverseCompleteSC} + \text{n} * T_{manual} \\ &> \text{n} * T_{traverseCompleteSC} \\ &> T_{traversePartSC\_product_1} + \dots + T_{traversePartSC\_product_n} \\ &= T_{generation\_c2} \end{aligned}$$

Therefore: $T_{generation\_c3} > T_{generation\_c2}$

$T_{generation\_c3} > T_{generation\_c2}$ shows that for the HMI development project from real life and based on the assumption that the chosen products to be tested together cover all provided functional features, functional feature options and variant features, the proposed approach modeling variability in the menu behavior SC saves generation time and hence achieves an improvement in efficiency.

9.2.3. General discussion

In the last subsection, it was mathematically shown that the proposed modeling and test generation approaches are efficient, based on the assumptions that 14 chosen products of a practical HMI project should be tested and that together they cover all provided functional features, functional feature options and variant features. In this section, we leave this concrete project and its assumptions and generally discuss the factors on which the improvement in efficiency is dependent.

The improvement in efficiency brought about by employing the proposed approaches is dependent on the following factors:

Factor 1 The number of layers in the menu behavior SC hierarchy

For our proof-of-concept test generator, we decided to use flattened SCs as explained at the end of Subsection 9.1.1. If the menu behavior SC is not flattened in advance, the test generation must resolve the same hierarchies every time the SC is traversed. In Methods 2 and 3, the SC traversal must be repeated for each product to be tested, whereas for Method 1 the traversal is done only once, regardless of the number of products to be tested. Therefore, if the used test generation is not based on flattened SCs, then the more hierarchy layers are used to describe the menu behavior, the more efficient is our generation approach.

Factor 2 Commonality of the menu behavior between the products

In Methods 2 and 3, the parts of the SC describing the common behavior of different products must be traversed for each product which is to be tested. That means the more similar the menu behaviors of different products are to each other, the more states must be redundantly visited for Methods 2 and 3 and consequently the more efficient are the proposed approaches.

Factor 3 The number of products to be tested

The test generation time of Method 1 does not increase with the number of products to be tested. In contrast, for Methods 2 and 3, the test generation must be performed as often as the number of products to be tested. At the end of the last subsection, it was shown that based on the practical HMI project, the test generation approach does not achieve any efficiency improvement if only one product of the PL is to be tested. However, for testing two or more products it is profitable to use the test generation method which avoids redundancies. This means that the more products there are to be tested, the better the efficiency of the proposed approaches.

Factor 4 The difference between the number of nodes in the generated test sequence graphs and the number of states in the SC

The test generation time of Method 1 comprises the time for traversing the SC and the time for traversing the relevant test sequence graphs. The advantage is that each test sequence graph is usually much smaller than the functional feature state from which it is generated. Each node of the test sequence graph "replaces" a sequence of states in the SC. Therefore the smaller the generated test sequence graphs are in comparison to their functional feature states, the more efficient the test generation method.

There can be cases in which the proposed test generation approach is not profitable, e.g. the case described at the end of Section 9.2.2 in which only one product is to be tested for the practical HMI development project. Generally, in cases where there is a disadvantageous combination of a very low number of products to be tested, a very low commonality between the behaviors of these products, and test sequence graphs which have sizes comparable to the functional feature states, the proposed generation method may be unprofitable. First, projects in which model-based testing approaches are applied usually have the goal of achieving a systematic and higher test coverage by testing as many products as possible. Second, it only makes sense to specify, develop and test the products as a product line if they have a large set of commonalities. Furthermore, it is unrealistic for a test sequence graph in practice to have as many nodes as the states in the respective functional feature state, since a menu behavior SC in practice does not contain so many variation points, junction points, breaking-function states or broken-function states which break a test sequence. This means that generated test sequences usually contain many steps and a test sequence graph is much smaller than the respective functional feature state. The proposed approach of modeling the menu behavior always achieves an improvement in efficiency due to the avoidance of invalid tests and manual work.

9.2.4. Estimation with concrete data

In this section we use some concrete data in order to show how much time can be saved by our approaches for the practical HMI development project. The time savings are estimated using data obtained from the proof-of-concept implementation and the practical HMI development model. Therefore, the results are strongly dependent on these data sources. They cannot serve as absolute statements but only provide an impression.

As explained above, 14 test benches are currently used in Audi's laboratory for function tests of the product line which we have introduced to be tested, and which provides 86 products. In order to provide a realistic impression, we first show the time saved for testing only 14 products and then for testing all 86 products.

The test generation time is strongly dependent on the number of states contained in the menu behavior SC, the processor and RAM of the PC on which the test generation is executed, and also the generator itself. To traverse our proof-of-concept menu behavior SC, which contains 14 view states, and to finish generating tests, our proof-of-concept test generator needs about 5 seconds. This is expressed as follows:

$T_{generation\_prototype} = \mathrm{t} = 5\mathrm{s}$

Method 1:

The menu behavior SC in the practical HMI development model contains 2349 view states (Table 9.8) which are about 2349 / 14 ≈ 168 times of the states in the proof-of-concept menu behavior SC. Since the searching time of depth search is linear, generating tests from the practical development model is approximately:

$T_{traverseCompleteSC} \approx 168 * T_{generation\_prototype} = 168 * \mathrm{t}$

As shown in Table 9.8, generated test sequence graphs contain about 15% nodes as the states in the functional feature states. Consequently:

$T_{traverseAllGraphs} \approx 15\% * T_{traverseCompleteSC} = 15\% * 168 * \mathrm{t} = 25.2 * \mathrm{t}$

Therefore, for Method 1:

$T_{generation\_c1} = T_{traverseCompleteSC} + T_{traverseAllGraphs} = 193.2 * \mathrm{t}$
with t = 5 s
$T_{generation\_c1} \approx$ **16 min**

The test generation time was estimated based on the measurement results of the proof-of-concept implementations and the numbers obtained from the real-life projects. This estimation shows that based on our proposed approaches, the test generation would take about 16 min in order to generate tests which cover all functional features, variant features and functional feature options on the same PC as the proof-on-concept test generation was executed. The generation time is independent of the number of products to be tested as long as all functional features, variant features and functional feature options shall be covered by the generated tests.

Method 2:

In the practical HMI development model, about 65% of the behavior describes the common behavior of different products. Consequently:

$$\begin{aligned} T_{traversePartSC\_product1} + ... + T_{traversePartSC\_product14} &\geq 14 * T_{traverse\_common} + T_{traverse\_specific} \\ &\approx 14 * 65\% * T_{traverseCompleteSC} + 35\% * T_{traverseCompleteSC} \\ &\approx 14 * 65\% * 168 * t + 35\% * 168 * t \\ &= 1587.6 * t \end{aligned}$$

Therefore, for Method 2:

$T_{generation\_c2} = T_{traversePartSC\_product\_1} + ... + T_{traversePartSC\_product\_14} = 1587.6 * t$

Consequently:
$T_{generation\_c2}$ **= 8.3 *** $T_{generation\_c1}$
And with t = 5 s:
$T_{generation\_c2} \approx$ **132.3 min**

For Method 2, the test generation has to be performed once for each product to be tested. The required generation time for Method 2 is about 8.3 times the generation time for Method 1. On the same PC as used for the first estimation, the test generation would take about 132.3 minutes. By applying the proposed test generation method, about 88% of the generation time can be saved.

Method 3:

For Method 3, even though we have no experience of the time demanded by the manual work, the time taken just to traverse the SC can already give a impression.

$14 * T_{traverseCompleteSC} \approx 14 * 168 * t = 2352 * t$

Therefore:

$T_{generation\_c3} = 14 * T_{traverseCompleteSC} + 14 * T_{manual} = 2352 * t + 7 * T_{manual}$

Consequently:
$T_{generation\_c3} >>$ **12 *** $T_{generation\_c1}$
with t = 5 s
$T_{generation\_c3} \approx$ **196 min + 7 *** T_{manual}

In this method, not only the test generation, but also the manual adaptation of generated tests has to be performed once for each product to be tested. Ignoring the time required for the manual work, the test generation alone for Method 3 required more than 3 hours which is more than 12 times the generation time for Method 1.

As discussed in the last section, the more products there are to be tested, the greater the difference between the test generation times for the three methods. In the following section, we perform the same estimations for the case in which all 86 products are to be tested.

The test generation time of Method 1 remains the same:

$T_{generation\_c1} = T_{traverseCompleteSC} + T_{traverseAllGraphs} = 193.2 * \text{t}$
with t = 5 s
$T_{generation\_c1} \approx$ **16 min**

For Method 2:

$T_{generation\_c2} = T_{traversePartSC\_product1} + ... + T_{traversePartSC\_product86} \approx 9450 * \text{t}$

Consequently:
$T_{generation\_c2} = \mathbf{49} * T_{generation\_c1}$
with t = 5 s
$T_{generation\_c2} \approx$ **13 h**

And for Method 3:

$T_{generation\_c3} = 86 * T_{traverseCompleteSC} + 86 * T_{manual} \approx 14448 * \text{t} + 86 * T_{manual}$

Consequently:
$T_{generation\_c3} >> \mathbf{75} * T_{generation\_c1}$
with t = 5 s
$T_{generation\_c3} \approx \mathbf{20\ h + 86} * T_{manual}$

The numbers above confirm our discussion above that the more products there are to be tested, the greater the efficiency improvement that can be achieved by applying our proposed approaches. In the case of testing all 86 available products, the test generation for Method 2 requires 13 hours, i.e. 49 times the generation time for Method 1 which remains the same despite the growing number of products to be tested. For Method 3, the test generation time alone requires 20 hours. If the time for manual adaptation is taken into account, the test generation effort would be extreme and unacceptable for serial HMI development.

9.3. Conclusions

In this chapter, the variability modeling and testing approaches have been evaluated from the efficiency aspect. We have first mathematically shown that the proposed approaches always achieve an efficiency improvement for real-life HMI projects. Afterwards, it was generally discussed on which factors the efficiency improvement is dependent. Also, worst cases in which the test generation approach is unprofitable were discussed. It was explained that such cases are rare for projects in which model-based testing approaches are applied. Finally, we demonstrated concrete data. In practice, it is usual to test only some available product variants in full. In our estimation, if only 14 of 86 products are to be tested, at least 88% of the generation time can be saved by applying our approaches. The difference between 16 minutes and 3 hours clearly demonstrates that our approaches can achieve an enormous improvement in efficiency for real-life projects. The case of testing all 86 products makes the benefits of our approaches even clearer; at least 98% of the generation time can be saved, i.e. about 13 hours.

Chapter 10.

Conclusion and Future Work

In this chapter, the contributions for this thesis will be summarized and an overview of future work will be given from the perspectives of industry and research.

10.1. Summary

The goal of this thesis is to provide a foundation for future HMI testing standards for the automotive industry and an approach for HMI testing under variability. In this section, we review and summarize the solutions proposed for these goals.

Currently, infotainment system HMI tests during serial development are performed al- most entirely manually. A model-based and automated HMI testing standard is still lacking. In particular, the features of the HMIs have not been sufficiently studied from the perspective of testing. For example, the types of HMI errors occurring in practice, their sources and the error distributions are still not documented in literature.

As preliminary work for this thesis, HMI errors occurring in practice have been analyzed and categorized based on the error sources. Figure 10.1 shows the types of HMI errors identified and their distribution.

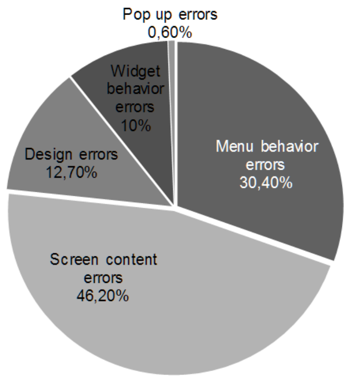


Figure 10.1.: HMI error distribution

As shown in the chart above, menu behavior errors and screen content errors make up the greater part of HMI errors. Menu behavior errors have a unique error source, i.e. the behavior model describing the dynamic menu behavior, whereas screen content errors can be grouped into several subclasses according to their different sources, which nonetheless are all in the implementation of the representation (screens, widgets etc.). In this thesis, the detection of menu behavior errors and screen content errors is defined as the focal point of the HMI testing framework.

The HMI testing framework proposed in this thesis is composed of four components, as shown in Figure 10.2.

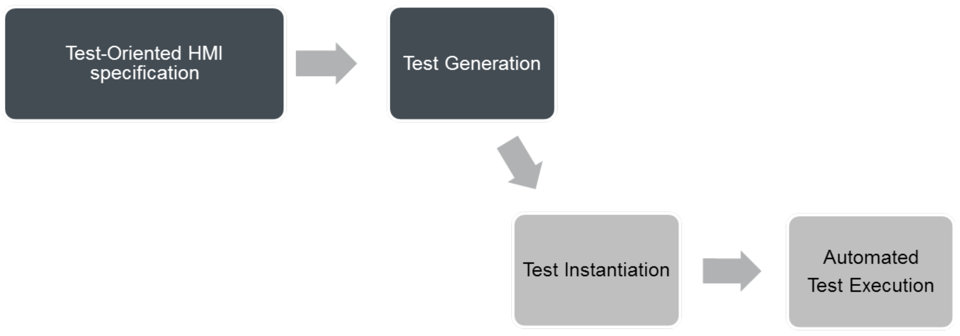


Figure 10.2.: The HMI testing framework

The test-oriented HMI specification is the most important component in the framework. It has a layered structure and describes the expectation of the HMI implementation to be tested. The test-oriented HMI specification contains sufficient data which is identified as necessary for performing menu behavior and screen content tests in a form which is suitable for test generation. The test generator automatically derives tests from the test-oriented HMI specification based on certain coverage criteria. For automatic test execution, generated tests must first be initialized and second, a test execution framework must be available which is suitable for HMI testing. The proposed HMI testing framework serves as a foundation for future HMI testing standards.

The variability of infotainment system HMIs leads to substantial testing efforts and hence poses a huge challenge to the HMI testing process. To address this, the software product line approach is applied in this thesis to integrate variability into the testing framework as follows: first, it is identified which types of variabilities can occur between practical HMI variants. The feature modeling technique is applied to describe both these variabilities and the commonalities on a high level. Second, the layers of the test-oriented HMI specification are extended to describe the commonalities and variabilities on a concrete level. Finally, methods are developed for the generation of tests for different products of a product line without having to redundantly repeat the generation process for each additional product. Using an algorithm, identical tests for different products can be identified and preserved from redundant executions.

By applying the model-based testing approach in the HMI testing framework, specific forms of test coverage can be achieved systematically. Automated test execution can greatly support foreign language system tests. During automated test executions, screenshots can be automatically captured of the screens visited; these screenshots can then be easily reviewed by native-speaker testers in order to verify the languages. In this way, it is not necessary to occupy test benches/cars or to manually execute the usually large number of tests. In particular, due to the proposed variability approaches, testing a large number of HMI variants can be much more efficient. First, the specification efforts can be reduced by avoiding redundant descriptions of commonalities. Second, the testing efforts can be reduced by avoiding redundant test generations and executions. The evaluation has shown that at least 88% of the test generation time can be saved on a practical HMI project by applying our generation approach.

At this point, we summarize the main contributions of this thesis:

- The HMI testing framework
  - serves as a foundation for future HMI testing standards for the automotive industry
  - supports abstract specifications of HMIs from test perspectives
  - supports automatic test generation
  - enables systematic test coverage
  - allows automatic test execution
- Extensions of the HMI testing framework for variability
  - provide an approach for testing complex and embedded GUIs/HMIs under variability
  - allow the specification of different HMI variants as a product line
  - allow the testing of different HMI variants as a product line
  - enable efficient test generation and avoid redundancies
  - significantly reduce the required effort

10.2. Future work

In order to apply the results of this thesis in practice, the testing concept can be further completed, refined and made more efficient.

Extension of the testing scope

The verification of menu behavior and screen content representation is defined as the focal point of the testing framework. As introduced in Section 4.2, the detection of several other error types can also be integrated into the test framework, i.e. **pop-up behaviors**, **widget behaviors** and **dynamic contents**. Also, design tests can be integrated into the framework. **Design tests** are used to verify whether design rules such as positions, color and fonts are observed in the HMI implementation. In a doctoral thesis currently in progress, design tests of infotainment system HMIs are investigated and the preliminary results documented in [50]. In that thesis, design tests require formal design information as a precondition. Due to the layered structure of our testing framework, design information can be specified in the design layer without having to adapt the current framework. Therefore, the results of that thesis can be considered to extend our testing framework for design tests. Recently, **animations** and **3D** representations have gradually been implemented in automotive HMIs. The testing of animations and 3D representations represents a major challenge for future HMI testing. Additionally, many errors can only be detected via **stress tests** which are usually performed by experienced testers in practice. Stress tests involve the whole infotainment system and more: the HMI, the instrument cluster, speech input and output facilities, underlying applications and bus systems. Activities which do not belong to infotainment system functions, e.g. electrically changing the seat setting, can also stress the behavior of the HMI. Integrating stress tests into the testing framework could be a huge challenge for future work but very effective. The testing framework can also be extended for **negative tests**, which e.g. avoid the occurrence of certain **scenarios**, **user inputs** or **exceeded reply time**. The generation of efficient **regression tests** for HMIs is also a very significant topic. For this, methods must be applied which are able to identify the "affected" tests in order to regenerate them exclusively. Finally, **multi-modality** has not been a focal point of this thesis. Testing the speech facilities and touch behaviors as well as the graphical behaviors of infotainment system HMIs based on model-based approaches is still a little-investigated field.

Intelligent and efficient testing

There is still great potential to improve the efficiency of the current testing approach, for which **Artificial Intelligence** (AI) methods can be considered. For instance, the number of generated HMI tests is still uncontrolled and usually very high. Many of the generated tests are very similar to each other in that they visit the same screens, have many overlapping test steps, require similar user inputs or produce similar outputs. According to the Pareto-Zipf–type laws ([31]), **reducing the number of generated tests** using an intelligent strategy can reduce testing efforts without affecting quality. Artificial Neural Network (ANN) has already been used for reducing tests generated from data-based models. By identifying input-output relationships based on the ANN, equivalent classes of generated tests can be determined. The goal is to select one or several representatives from each class. HMI models are usually event-based models. ANN or similar AI methods need to be extended for event-based models and especially HMI tests. For instance, it is conceivable to define relationships for screens based on certain occurring orders and identify screen relationships in order to identify equivalent classes. The same idea can also be applied for user actions. The quality of generated tests still has scope for improvement. For instance, in order to select test cases having higher probabilities of error detection, AI methods such as AI planning [53] and generic methods [102] can be considered. This would be a very attractive topic, since HMI testing in serial development usually suffers from a lack of time.

Adequate coverage criteria for HMI tests

In this thesis, test generation has been mainly based on the conventional structural coverages. Further **coverage criteria** still need to be defined which are especially adequate for HMI testing.

An infotainment system provides numerous functions. A user generally uses several of them much more frequently than others. Similarly, the HMI usually provides several options for the user to access the same function, e.g. in order to start the navigation guidance, the user can input a city followed by a street or choose the center of the city. The user usually employs one or some of these options much more frequently than others. This means that different functions and scenarios have higher priorities for testing. This fact should be better considered in future HMI testing. In order to generate tests according to these priorities, a kind of "**usage-oriented coverage**" could be defined and applied. For instance, the priorities of screens, transitions and/or transition sequences could be labeled with properties . Different levels of tests could be generated based on these priorities. For instance, level I contains tests verifying the correct start-up of the system and the accessibility of the functions, while level II contains tests verifying the basic functions or the most used functions etc. It could be even more interesting to define several different "usage profiles" for different testing phases and testing goals. For instance, one usage profile is defined for generating minimal tests, one for stress tests and one for random tests. One could imagine that users in Asian or Arabian counties interact differently with the HMI than users in European countries. Therefore, usage profiles can also be defined for market-specific tests. Up to now we have only focused on coverage criteria for testing the dynamic behavior of HMIs. Coverage criteria still need be defined for other types of tests, e.g. for design tests it must be defined which types of widgets and which of their properties are to be tested.

Until now we are only focused on coverage criteria for testing the dynamic behavior of HMIs until now. Coverage criteria still need be defined for other types of tests, e,g. for design tests it must be defined which types of widgets and which properties of them are to be tested.

Adaptive HMI testing

Future HMI testing can be made more **adaptive** to improve testing efficiency. For example, in connection with regression tests, the testing concept must ensure that scenarios which have been verified as erroneous in one of the previous HMI versions must be retested for the next HMI version. This requires traceability from a generated test to the scenario described in the specification. The scenario can be traced, for example, by marking the screens, widgets, user actions and/or user action sequences. For the next HMI version to be tested, if it is detected that changes have occurred in the traced scenario, then one or several new tests should be generated from the specification which corresponds to the HMI version to be tested. Otherwise, the old tests can be reused for retesting. In order to "learn" from past errors and ensure they are retested, a kind of "**past error coverage**" can be considered which defines which past errors must be covered by the tests. Ideally, "past error coverage" can be used in combination with a variable which defines up to which past version errors should be"retested" for the current version. Also, scenarios which are verified to be less error-prone or never error-prone in past tests can be considered for omission from certain test phases. Maybe it is also possible to learn from early errors and create new tests which have a higher probability of detecting them. Furthermore, generated tests still cannot achieve the quality of testing by experienced test experts who usually find more complicated errors. It would also be a very attractive research area to learn from human testers and generate more intelligent HMI tests. The scope for making the HMI testing more adaptive is still wide.

Avoiding errors during specification construction

Model-based testing approaches imply the problem that if **errors exist in the specification**, the SUT is verified against erroneous tests. This leads to “falsified” testing results and hence unnecessary effort for error logging. In future HMI testing, errors shall ideally be detected and avoided during the construction of specifications.

Many modeling errors can be determined on the tool level. For instance, trivial errors such as unreachable states and duplicated transitions can already be automatically reported by most of the HMI development tools during specification. Future HMI specification tools should better determine more complex modeling errors such as conflicts in the conditions or HMI-specific errors such as an outgoing transition starting from a view state of which the associated screen cannot trigger the event of this transition.

Errors can also occur during specification which are not so trivial, e.g. inconsistency of the operation concept. For instance, in the operation concept for one market, it is defined that “return” buttons always lead the user to the previous screen visited by the user, whereas for another market, it is defined that “return” buttons always lead the user to the next screen up in the screen hierarchy. Mistakes can be made with the operation concepts, which leads to inconsistency in the specification. In order to avoid these types of errors, **patterns** can be considered. For instance, for different markets, patterns for the “return” concept can be predefined which must be observed in the whole specification of the corresponding market.

Usability testing for HMIs

Usability is still a very complex and hot topic in the area of user interfaces. A number of user studies are being conducted by various manufacturers. Although there are perhaps no GUIs which have absolutely perfect usability, several heuristics and standards are however accepted from which metrics could be derived for usability measurement. Such metrics could be used for automated and model-based usability testing of infotainment system HMIs, which is still an open area and a very interesting research topic. For instance, a Japanese guideline for infotainment system HMIs defines that a function which requires more than four user actions has a low level of usability and must be deactivated during driving. Such usability problems can be precluded in the specification phase, e.g. by extending specification tools with constraints limiting the number of allowed user actions in order to access a screen from the initial screen or the number of screen hierarchies.

Application in practice

In order to apply the results of this thesis in serial development, manufacturers’ HMI development processes may need to be adapted. The proposed approach requires a compatible development process, capable teams and suitable tools. For instance, many manufacturers have not directly integrated the development of foreign language system HMIs into the development process for their home market. Either there are no explicit specifications for foreign HMIs or the deviations from the HMIs for the home market are described in a separate document. Development processes need be adapted or optimized for applying the proposed approach. Additionally, capable teams which are able to, for example, extend the HMI specification with the required test information, are necessary. In addition, suitable tools must be available, which, for example, allow abstract and uncompleted HMI specifications.

10.3. Final words

Although the testing framework can and needs to be further refined and made more efficient, the first fundamentals have been provided by this thesis for HMI testing under variability. For the first time, HMI errors occurring in practice have been studied and categorized. The HMI testing framework is based on the goal of efficiently and independently detecting these errors with as little specification efforts as possible. Furthermore, HMI variability with real-life complexity has been analyzed and integrated into the testing framework. The proposed variability specification and testing approaches have been evaluated to be extremely time-saving. The evaluation in Chapter 9 has shown that at least 88% of the generation time can be saved by applying the variability solutions for a practical HMI project. We are convinced that based on the results of this thesis, a systematic test coverage and hence better quality can be achieved with much lower effort and costs for future HMI testing and especially for multi-variant HMI testing.

Part V.

Bibliography

Bibliography

[1] de Almeida, E. S. (Hrsg.) ; Kishi, T. (Hrsg.) ; Schwanninger, C. (Hrsg.) ; John, I. (Hrsg.) ; Schmid, K. (Hrsg.): *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011.* IEEE, 2011 . – ISBN 978–1–4577–1029–2

[2] Alsmadi, I. M.: *Building a user interface test automation framework using the data model.* Fargo, ND, USA, Diss., 2008. – AAI3305533

[3] Ambler, S. W.: *The Elements of UML(TM) 2.0 Style.* New York, NY, USA : Cambridge University Press, 2005. – ISBN 0521616786

[4] Ammann, P. ; Offutt, J. ; Xu, W. : Formal methods and testing. Berlin, Heidelberg : Springer-Verlag, 2008. – ISBN 3–540–78916–2, 978–3–540–78916–1, Kapitel Coverage criteria for state based specifications, S. 118–156

[5] Anthonysamy, P. ; Somé, S. S.: *Aspect-oriented use case modeling for software product lines.* ACM Press, 2008, S. 1–8

[6] Antkiewicz, M. ; Czarnecki, K. : FeaturePlugin: feature modeling plug-in for Eclipse. In: Burke, M. G. (Hrsg.): *ETX*, ACM, 2004, S. 67–72

[7] AUDI AG and BMW and Daimler AG and Porsche and VW: *XML-Sprache zur Beschreibung von HMIs für Infotainmentsysteme und Kombiinstrumente.* 2007. – Internal document of project automotiveHMI

[8] Bagheri, E. ; Di Noia, T. ; Ragone, A. ; Gasevic, D. : Configuring software product line feature models based on Stakeholders' soft and hard requirements. In: *Proceedings of the 14th international conference on Software product lines: going beyond.* Berlin, Heidelberg : Springer-Verlag, 2010 (SPLC'10). – ISBN 3–642–15578–2, 978–3–642–15578–9, S. 16–31

[9] Baker, P. ; Dai, Z. R. ; Grabowski, J. ; Haugen, y. ; Schieferdecker, I. ; Williams, C. : *Model-Driven Testing: Using the UML Testing Profile.* 1. Springer, Berlin, 2007. – ISBN 3540725628

[10] Bauer, S. : *Implementierung praktischer Anwendung zur Unterstuetzung der Testautomatisierung von Automotive Infotainmentsystemen basierend auf der Analyse des Human-Machine-Interface*, Ludwig-Maximilians-Universität München, Diploma thesis, 2011

[11] von der Beeck, M. : A structured operational semantics for UML-statecharts. In: *Software and Systems Modeling* V1 (2002), Dezember, Nr. 2, S. 130–141. – ISSN 1619–1366

[12] Belli, F. : Finite-State Testing and Analysis of Graphical User Interfaces. In: *ISSRE '01: Proceedings of the 12th International Symposium on Software Reliability Engineering.* Washington, DC, USA : IEEE Computer Society, 2001. – ISBN 0–7695–1306–9, S. 34

[13] Benz, S. : *Generating tests for feature interaction*, Technical University Munich, Diss., 2010

[14] Berti, S. ; Correani, F. ; Mori, G. ; Paternò, F. ; Santoro, C. ; Moruzzi, V. G.: TERESA: A Transformation-based Environment for Designing and Developing Multi-Device Interfaces. In: *Evaluation* (2004), S. 1–2

[15] Books, H. : *UML Diagrams, Including: State Diagram, Sequence Diagram, Class Diagram, System Sequence Diagram, Deployment Diagram, Package Diagram, Component Diagram, Activity Diagram, Communication Diagram, Timing Diagram (Unified Modeling Language), Object Diagrams.* 1st. Hephaestus Books, 2011. – ISBN 0321635841, 9780321635846

[16] Born, M. ; Ekine, O. ; Ashley, J. : A model-driven product line approach for HMI development. In: *Tagungsband der PIK 2009-Produktlinien im Kontext* Bd. 111, 2009, S. 26–33

[17] Bullard, V. ; Smith, K. ; Daconta, M. : *Essential XUL programming.* Wiley, 2001 (Wiley computer publishing). – ISBN 9780471415800

[18] Camastra, F. : Image Processing: Principles and Applications [book review]. In: *IEEE Transactions on Neural Networks* 18 (2007), Nr. 2, S. 610

[19] Chang, T.-H. ; Yeh, T. ; Miller, R. C.: GUI testing using computer vision. In: *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems.* New York, NY, USA : ACM, 2010. – ISBN 978–1–60558–929–9, S. 1535–1544

[20] Chinnapongse, V. ; Lee, I. ; Sokolsky, O. ; Wang, S. ; Jones, P. L.: Model-Based Testing of GUI-Driven Applications. In: *Software Technologies for Embedded and Ubiquitous Systems* Bd. 5860/2009. Springer-verlag New York Inc, 2009. – 7th IFIP WG 10.2 International Workshop, SEUS 2009 Newport Beach, CA, USA, 2009 Proceedings, S. 203–214

[21] Claude, G. ; Ruediger, V. : *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications.* Springer Berlin Heidelberg, 2002

[22] Clauss, M. : Generic Modeling using UML extensions for variability. In: *Proceedings of OOPSLA Workshop on Domain-specific Visual Languages.* Tampa, FL, USA, 2001, S. 11–18

[23] Clements, P. ; Northrop, L. : *Salion, Inc: a software product line case study.* Carnegie Mellon University, Software Engineering Institute, 2002 (Technical report)

[24] Czarnecki, K. : Mapping features to models: A template approach based on superimposed variants. In: *GPCE 2005 - Generative Programming and Component Engineering. 4th International Conference*, Springer, 2005, S. 422–437

[25] Czarnecki, K. ; Eisenecker, U. W.: *Generative programming: methods, tools, and applications*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 2000. – ISBN 0–201–30977–7

[26] Drusinsky, D. : *Modeling and Verification Using UML Statecharts, First Edition : A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*. Newnes, April 2006. – ISBN 0750679492

[27] Duan, L. ; Hoefer, A. ; Hussmann, H. : 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, 2010

[28] Duan, L. ; Hussmann, H. ; Hoefer, A. : A Test-Oriented HMI Specification Model for Model-Based Testing of Automotive Human-Machine Interfaces. In: Fähnrich, K.-P. (Hrsg.) ; Franczyk, B. (Hrsg.): *GI Jahrestagung (2)* Bd. 176, GI, 2010. – ISBN 978–3–88579–270–3, S. 339–344

[29] Duan, L. ; Hussmann, H. ; Niederkorn, D. ; Hoefer, A. : Model-Based Testing for the Menu Behavior of Automotive Infotainment System HMIs. In: *Proceedings of the 1st International Workshop on Model-based Interactive Ubiquitous Systems 2011*. Pisa, Italy : CEUR, 2011, S. 15–20

[30] Education, L. O.: Model View Controller Design Pattern. In: *Online* 3 (2007), Nr. Wong 2003, S. 2000–2007

[31] Endres, A. ; Rombach, D. : *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. illustrated. Addison Wesley, Mai 2003. – ISBN 0321154207

[32] Feldt, K. : *Programming Firefox: Building Rich Internet Applications with Xul*. O'Reilly Media, Inc., 2007. – ISBN 0596102437

[33] Fenton, N. E. ; Ohlsson, N. : Quantitative Analysis of Faults and Failures in a Complex Software System. In: *IEEE Trans. Softw. Eng.* 26 (2000), August, S. 797–814. – ISSN 0098–5589

[34] Fernandes, P. ; Werner, C. ; Teixeira, E. : An Approach for Feature Modeling of Context-Aware Software Product Line. In: *Journal of Universal Computer Science* 17 (2011), Mar, Nr. 5, S. 807–829

[35] Fewster, M. ; Graham, D. : *Software test automation: effective use of test execution tools*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1999. – ISBN 0–201–33140–3

[36] Filip, J. ; Haindl, M. : Bidirectional Texture Function Modeling: A State of the Art Survey. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 31 (2009), November, S. 1921–1940. – ISSN 0162–8828

[37] Fleischmann, T. : Model Based HMI Specification in an Automotive Context. In: *Human Interface and the Management of Information. Methods, Techniques and Tools in Information Design* Bd. 4557/2007. Springer Berlin / Heidelberg, 2007, S. 31–39

[38] Fröhlich, P. ; Link, J. : Automated Test Case Generation from Dynamic Models. In: *Proceedings of the 14th European Conference on Object-Oriented Programming*. London, UK : Springer-Verlag, 2000 (ECOOP '00). – ISBN 3–540–67660–0, S. 472–492

[39] Gacek, C. ; Anastasopoules, M. : Implementing product line variabilities. In: *SIGSOFT Softw. Eng. Notes* 26 (2001), May, S. 109–117. – ISSN 0163–5948

[40] Ganov, S. R. ; Killmar, C. ; Khurshid, S. ; Perry, D. E.: Test generation for graphical user interfaces based on symbolic execution. In: *Proceedings of the 3rd international workshop on Automation of software test*. New York, NY, USA : ACM, 2008 (AST '08). – ISBN 978–1–60558–030–2, S. 33–40

[41] García, J. G. ; Calleros, J. M. G. ; Vanderdonckt, J. ; Arteaga, J. M.: A Theoretical Survey of User Interface Description Languages: Preliminary Results. In: *LA-WEB/CLIHC*, IEEE Computer Society, 2009. – ISBN 978–0–7695–3856–3, S. 36–43

[42] Gaston, C. ; Seifert, D. : Evaluating Coverage Based Testing. In: *Model-Based Testing of Reactive Systems*. Springer-Verlag New York, LLC, 2005

[43] Gnesi, S. ; Latella, D. ; Massink, M. : Formal Test-Case Generation for UML Statecharts. In: *Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7695–2109–6, S. 75–84

[44] Gonzalez, A. ; Luna, C. : Behavior Specification of Product Lines via Feature Models and UML Statecharts with Variabilities. In: *Proceedings of the 2008 International Conference of the Chilean Computer Science Society*. Washington, DC, USA : IEEE Computer Society, 2008. – ISBN 978–0–7695–3403–9, S. 32–41

[45] Grandy, H. ; Benz, S. : Specification based testing of automotive human machine interfaces. In: *GI Jahrestagung'09* Bd. 154, GI, 2009, S. 2720–2727

[46] Gross, A. ; Eisenbarth, M. ; Naegele, F. ; Klaus, A. ; Maier, A. ; Orfgen, M. ; Kuemmerling, M. : *HMI-Entwicklungsprozesse*. Research project automotiveHMI, 2011. – Internal document

[47] Grzemba, A. : *MOST The Automotive Multimedia Network*. Franzis, 2008

[48] Halmans, G. ; Pohl, K. : Communicating the variability of a software-product family to customers. In: *Informatik - Forschung und Entwicklung* 18 (2004), April, Nr. 3, S. 113–131

[49] Hammontree, M. L. ; Hendrickson, J. J. ; Hensley, B. W.: Integrated data capture and analysis tools for research and testing on graphical user interfaces. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM, 1992 (CHI '92). – ISBN 0–89791–513–5, S. 431–432

[50] Heilemann, M. ; Palm, G. : Automatisierte Verifikation des Designs der grafischen Benutzeroberfläche von Infotainmentsystemen mit einem Bayes'schen Netzwerk. In: *INFORMATIK 2011 Informatik schafft Communities Proceedings der 41. GI-Jahrestagung*, 2011, S. 212–227

[51] Hong, H. S. ; Kim, Y. G. ; Cha, S. D. ; Bae, D. H. ; Ural, H. : A Test Sequence Selection Method for Statecharts 1 Introduction. In: *Science And Technology* 10 (2000), Nr. 4, S. 203–227

[52] Hong, H. S. ; Lee, I. ; Sokolsky, O. ; Cha, S. D.: Automatic Test Generation from Statecharts Using Model Checking. In: *BRICS, Proceedings of the Workshop on Formal Approaches to Testing of Software, FATES'01g.* Aalborg, Denmark, 2001, S. 15–30

[53] Howe, A. E. ; Mayrhauser, A. v. ; Mraz, R. T.: Test Case Generation as an AI Planning Problem. In: *Automated Software Engg.* 4 (1997), January, S. 77–106. – ISSN 0928–8910

[54] Huang, Y. ; McMurran, R. ; Amor-Segan, M. ; Dhadyalla, G. ; Jones, R. P. ; Bennett, P. ; Mouzakitis, A. ; Kieloch, J. : Development of an automated testing system for vehicle infotainment system. In: *The International Journal of Advanced Manufacturing Technology* (2010), S. 1–14

[55] Jacky, J. ; Veanes, M. ; Campbell, C. ; Schulte, W. : *Model-Based Software Testing and Analysis with C#.* 1. Cambridge University Press, 2007. – ISBN 0521687616, 9780521687614

[56] John, I. ; Muthig, D. : Tailoring Use Cases for Product Line Modeling. In: *Proceedings of the International Workshop on Requirements Engineering for Product Lines*, 2002, S. 26–32

[57] Jones, J. P.: A concurrent on-board vision system for a mobile robot. In: *Proceedings of the third conference on Hypercube concurrent computers and applications - Volume 2.* New York, NY, USA : ACM, 1988 (C3P). – ISBN 0–89791–278–0, S. 1022–1032

[58] Jéron, T. ; Morel, P. : Test Generation Derived from Model-Checking. In: Halbwachs, N. (Hrsg.) ; Peled, D. (Hrsg.): *CAV* Bd. 1633, Springer, 1999. – ISBN 3–540–66202–2, S. 108–121

[59] Kang, K. C. ; Lee, J. ; Donohoe, P. : Feature-oriented product line engineering. In: *Software, IEEE* 19 (2002), Juli, Nr. 4, S. 58–65. – ISSN 0740–7459

[60] Kasik, D. J. ; George, H. G.: Toward automatic generation of novice user test scripts. In: *Proceedings of the SIGCHI conference on Human factors in computing systems: common ground.* New York, NY, USA : ACM, 1996 (CHI '96). – ISBN 0–89791–777–4, S. 244–251

[61] Kasurinen, J. ; Taipale, O. ; Smolander, K. : Software test automation in practice: empirical observations. In: *Adv. Soft. Eng.* 2010 (2010), January, S. 4:1–4:13. – ISSN 1687–8655

[62] Kervinen, A. : *Towards practical model-based testing: improvements in modelling and test generation.* Tampere University of Technology, 2008. – ISBN 9789521520655

[63] Kervinen, A. ; Maunumaa, M. ; Pääkkönen, T. ; Katara, M. : Model-based testing through a GUI. In: *Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software (FATES 2005), number 3997 in Lecture Notes in Computer Science*, Springer, 2006, S. 16–31

[64] Kiffe, G. : *EXtended Automation Method (EXAM) Konzeptpapier Version 3.0.* Ingolstadt: Audi AG, 2010. – Internal document

[65] Kuemmerling, M. ; Meixner, G. : Modellbasiertes Austauschformat zur optimierten HMI-Entwicklung. In: *ATZ elektronik* (2011)

[66] Lamancha, B. P. ; Usaola, M. P. ; Velthius, M. P.: Software Product Line Testing - A Systematic Review. In: *ICSOFT (1).* Sofia, Bulgaria : INSTICC Press, 2009. – ISBN 978–989–674–009–2, S. 23–30

[67] Lee, D. ; Yannakakis, M. : Principles and methods of testing finite state machines - a survey. In: *Proceedings of the IEEE* 84 (1996), Nr. 8, S. 1090–1123

[68] Lee, K. ; Kang, K. C. ; Lee, J. : *Concepts and Guidelines of Feature Modeling for Product Line Software Engineering.* Bd. LNCS 2319. Springer, 2002, S. 62–77

[69] Leffingwell, D. : *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise.* 1st. Addison-Wesley Professional, 2011. – ISBN 0321635841, 9780321635846

[70] Legeard, B. ; Peureux, F. ; Utting, M. : Controlling test case explosion in test generation from B formal models: Research Articles. In: *Softw. Test. Verif. Reliab.* 14 (2004), June, S. 81–103. – ISSN 0960–0833

[71] Limbourg, Q. ; Vanderdonckt, J. ; Michotte, B. ; Bouillon, L. ; Lopez-Jaquero, V. : UsiXML: A Language Supporting Multi-path Development of User Interfaces. Berlin, Heidelberg : Springer Berlin / Heidelberg, 2005, Kapitel 12, S. 134–135

[72] M., M. A. ; E., P. M. ; Lou, S. M.: Hierarchical GUI Test Case Generation Using Automated Planning. In: *IEEE Trans. Softw. Eng.* 27 (2001), Nr. 2, S. 144–155. – ISSN 0098–5589

[73] MacVittie, L. A.: *XAML in a Nutshell (In a Nutshell (O'Reilly)).* O'Reilly Media, Inc., 2006. – ISBN 0596526733

[74] von der Massen, T. ; Lichter, H. : Modellierung von Variabilität mit UML Use Cases. In: *Softwaretechnik-Trends* 23 (2003), Nr. 1

[75] Marciniak, J. J.: *Encyclopedia of Software Engineering.* 2nd. New York, NY, USA : John Wiley & Sons, Inc., 2002. – ISBN 0471210072

[76] Mc Quillan, J. A. ; Power, J. F.: A Survey of UML-Based Coverage Criteria for Software Testing / Department of Computer Science. Maynooth, Co. Kildare, Ireland, 2005. – Forschungsbericht

[77] Meixner, G. : *Entwicklung einer modellbasierten Architektur für multimodale Benutzungsschnittstellen*, Fachbereich Maschinenbau und Verfahrenstechnik, TU Kaiserslautern, PhD-Thesis, February 2010. – 201 S

[78] Memon, A. M.: Advances in GUI Testing. In: Zelkowitz, M. V. (Hrsg.): *Highly Dependable Software – Advances in Computers* Bd. 58. Academic Press, 2003, S. 149–201

[79] Memon, A. M.: An event-flow model of GUI-based applications for testing. In: *Software Testing Verification and Reliability* 17 (2007), Nr. 3, S. 137–157

[80] Memon, A. M. ; Pollack, M. E. ; Soffa, M. L.: Using a goal-driven approach to generate test cases for GUIs. In: *Proceedings of the 21st international conference on Software engineering*. New York, NY, USA : ACM, 1999 (ICSE '99). – ISBN 1–58113–074–0, S. 257–266

[81] Memon, A. M. ; Pollack, M. E. ; Soffa, M. L.: Plan Generation for GUI Testing. In: *Proceedings of The Fifth International Conference on Artificial Intelligence Planning and Scheduling*, AAAI Press, April 2000, S. 226–235

[82] Memon, A. M. ; Soffa, M. L.: Regression testing of GUIs. In: *Proceedings of the 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, ACM Press, 2003, S. 118–127

[83] Memon, A. M. ; Soffa, M. L. ; Pollack, M. E.: Coverage criteria for GUI testing. In: *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering ESECFSE9* 26 (2001), Nr. 5, S. 256

[84] Memon, A. M.: *A comprehensive framework for testing graphical user interfaces*, Diss., 2001. – AAI3026063

[85] Mueller, W. ; Schaefer, R. ; Bleul, S. : Interactive Multimodal User Interfaces for Mobile Devices. In: *Proceedings of Hawaii International Conference on System Sciences, HICSS 37*, 2004, S. 5–8

[86] Müller-Bagehl, C. (Hrsg.) ; Endt, P. (Hrsg.): *Haus der Technik Fachbuch*. Bd. 38: *Infotainment/Telematik im Fahrzeug: Trends für die Serienentwicklung*. Renningen : Expert, 2004. – ISBN 3–816–92440–9

[87] Neto, A. C. D. ; Travassos, G. H.: Model-based testing approaches selection for software projects. In: *Information & Software Technology* 51 (2009), Nr. 11, S. 1487–1504

[88] Nguyen, D. H. ; Strooper, P. ; Suess, J. G.: Model-based testing of multiple GUI variants using the GUI test generator. In: *Proceedings of the 5th Workshop on Automation of Software Test*. New York, NY, USA : ACM, 2010 (AST '10). – ISBN 978–1–60558–970–1, S. 24–30

[89] Nikolai, A. P. ; Paiva, A. C. R. ; Tillmann, N. ; Faria, J. C. P. ; M, R. F. A.: Modeling and Testing Hierarchical GUIs. In: *Proc.ASM05. Université de Paris 12*, 2005, S. 8–11

[90] Offutt, J. ; Abdurazik, A. : Generating Tests from UML Specifications. In: France, R. (Hrsg.) ; Rumpe, B. (Hrsg.): *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings* Bd. 1723, Springer, 1999, S. 416–429

[91] Offutt, J. ; Liu, S. ; Abdurazik, A. ; Ammann, P. : Generating test data from state-based specifications. In: *The Journal of Software Testing, Verification and Reliability* 13 (2003), S. 25–53

[92] Oldevik, J. : Can aspects model product lines? In: *Proceedings of the 2008 AOSD workshop on Early aspects.* New York, NY, USA : ACM, 2008 (EA '08). – ISBN 978–1–60558–143–9, S. 3:1–3:8

[93] Olimpiew, E. M.: *Model-based testing for software product lines.* Fairfax, VA, USA, Diss., 2008. – AAI3310145

[94] Olimpiew, E. M. ; Gomaa, H. : Model-based testing for applications derived from software product lines. In: *Proceedings of the 1st international workshop on Advances in model-based testing.* New York, NY, USA : ACM, 2005 (A-MOST '05). – ISBN 1–59593–115–5, S. 1–7

[95] Paiva, A. C. R. ; Faria, J. C. P. ; Vidal, R. F. A. M.: Towards the Integration of Visual and Formal Models for GUI Testing. In: *Electr. Notes Theor. Comput. Sci.* 190 (2007), Nr. 2, S. 99–111

[96] Paiva, A. C. R. ; Faria, J. C. P. ; Tillmann, N. ; Vidal, R. F. A. M.: A Model-to-Implementation Mapping Tool for Automated Model-Based GUI Testing. In: Lau, K.-K. (Hrsg.) ; Banach, R. (Hrsg.): *ICFEM* Bd. 3785, Springer, 2005. – ISBN 3–540–29797–9, S. 450–464

[97] Palanque, P. A. ; Bastide, R. : Petri net based Design of User-driven Interfaces Using the Interactive Cooperative Objects Formalism. In: Paternò, F. (Hrsg.): *Design, Specification and Verification of Interactive Systems 94, Proceedings of the First International Eurographics Workshop, June 8-10, 1994, Bocca di Magra, Italy*, Springer, 1994. – ISBN 3–540–59480–9, S. 383–400

[98] Paterno', F. ; Santoro, C. ; Spano, L. D.: MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. In: *ACM Trans. Comput.-Hum. Interact.* 16 (2009), November, Nr. 4, S. 1–30

[99] Phanouriou, C. : *UIML: A Device-Independent User Interface Markup Language*, Citeseer, Diss., 2000. – 161 S

[100] Pleuss, A. ; Hussmann, H. : Model-Driven Development of Interactive Multimedia Applications with MML. In: Hussmann, H. (Hrsg.) ; Meixner, G. (Hrsg.) ; Zuehlke, D. (Hrsg.): *Model-Driven Development of Advanced User Interfaces* Bd. 340. Springer Berlin / Heidelberg, 2011, S. 199–218

[101] Pohl, K. ; Böckle, G. ; Van Der Linden, F. : *Software product line engineering: foundations, principles, and techniques.* Springer, 2005. – 467 S

[102] Popovic, M. ; Velikic, I. : A Generic Model-Based Test Case Generator. In: *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems.* Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2308–0, S. 221–228

[103] Reis, S. ; Pohl, K. : Wiederverwendung von Integrationstestfällen in der Software-Produktlinienentwicklung. In: *Inform., Forsch. Entwickl.* 22 (2008), Nr. 4, S. 267–283

[104] Reza, H. ; Endapally, S. ; Grant, E. : A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets. In: *Proceedings of the International Conference on Information Technology.* Washington, DC, USA : IEEE Computer Society, 2007 (ITNG '07). – ISBN 0–7695–2776–0, S. 366–370

[105] Rossner, T. (Hrsg.) ; Brandes, C. (Hrsg.) ; Goetz, H. (Hrsg.) ; Winter, M. (Hrsg.): *Basiswissen-Modellbasierter Test.* Bd. 424. dpunkt.verlag, 2010

[106] Samek, M. : *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems.* 2. Newton, MA, USA : Newnes, 2008. – ISBN 0750687061, 9780750687065

[107] Schwarzl, C. ; Peischl, B. : Test Sequence Generation from Communicating UML State Charts: An Industrial Application of Symbolic Transition Systems. In: *Proceedings of the 2010 10th International Conference on Quality Software.* Washington, DC, USA : IEEE Computer Society, 2010 (QSIC '10). – ISBN 978–0–7695–4131–0, S. 122–131

[108] Shehady, R. K. ; Siewiorek, D. P.: 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).* Washington, DC, USA : IEEE Computer Society, 1997 (FTCS '97). – ISBN 0–8186–7831–3, S. 80–

[109] Smaragdakis, Y. ; Batory, D. : Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. In: *ACM Trans. Softw. Eng. Methodol.* 11 (2002), April, S. 215–255. – ISSN 1049–331X

[110] Souchon, N. ; Vanderdonckt, J. : A review of XML-compliant user interface description languages. In: *Lecture notes in computer science* 2844 (2003), S. 377–391

[111] Stoiber, R. ; Meier, S. ; Glinz, M. : Visualizing Product Line Domain Variability by Aspect-Oriented Modeling. In: *Proceedings of the Second International Workshop on Requirements Engineering Visualization.* Washington, DC, USA : IEEE Computer Society, 2007 (REV '07). – ISBN 0–7695–3248–9, S. 8–

[112] Stolle, R. ; Benedek, T. ; Knuechel, C. : Model-Based Test Automation for Automotive HMIs. In: *Jahrestagung der ASIM/GI-Fachgruppe 4.5.5 Simulation technischer Systeme, Simulations- und Testmethoden fuer Software in Fahrzeugsystemen, ISSN 1436-9915.* Berlin, Germany, 2005

[113] Stricker, V. ; Metzger, A. ; Pohl, K. : Avoiding redundant testing in application engineering. In: *Proceedings of the 14th international conference on Software product lines: going beyond.* Berlin, Heidelberg : Springer-Verlag, 2010 (SPLC'10). – ISBN 3–642–15578–2, 978–3–642–15578–9, S. 226–240

[114] Szasz, N. ; Vilanova, P. : Statecharts and Variabilities. In: Heymans, P. (Hrsg.) ; Kang, K. C. (Hrsg.) ; Metzger, A. (Hrsg.) ; Pohl, K. (Hrsg.) ; Heymans, P. (Hrsg.) ; Kang, K. C. (Hrsg.) ; Metzger, A. (Hrsg.) ; Pohl, K. (Hrsg.): *VaMoS*, 2008 (ICB Research Report), S. 131–140

[115] Tawhid, R. ; Petriu, D. C.: Automatic Derivation of a Product Performance Model from a Software Product Line Model. In: *SPLC*, 2011, S. 80–89

[116] Thiel, S. ; Hein, A. : Modeling and Using Product Line Variability in Automotive Systems. In: *IEEE Softw.* 19 (2002), July, S. 66–72. – ISSN 0740–7459

[117] Voelter, M. ; Groher, I. : Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: *Proceedings of the 11th International Software Product Line Conference*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2888–0, S. 233–242

[118] Weber, M. ; Berton, A. ; Reich, B. : Abstrakte Beschreibung automobiler HMI-Systeme (Abstract Description of Automobile HMI systems). In: *i-com* 8 (2009), Nr. 2, S. 15–18

[119] Wegner, G. ; Endt, P. ; Angelski, C. : Das elektrische Lastenheft als Mittel zur Kostenreduktion bei der Entwicklung der Menschen-Machine-Schnittstelle von Infotinament-Systemen im Fahrzeug. In: *Infotainment Telematik im Fahrzeug*. Expert-Verlag GmbH, 2004, S. 38–45

[120] Weissleder, S. ; Sokenou, D. ; Schlinglo, B.-H. : Reusing State Machines for Automatic Test Generation in Product Lines. In: *1st Workshop on Model-based Testing in Practice (MoTiP 2008)*. Berlin, Germany, Juni 2008

[121] White, L. ; Almezen, H. : Generating Test Cases for GUI Responsibilities Using Complete Interaction Sequences. In: *Proceedings of the 11th International Symposium on Software Reliability Engineering*. Washington, DC, USA : IEEE Computer Society, 2000 (ISSRE '00). – ISBN 0–7695–0807–3, S. 110–

[122] White, L. ; Almezen, H. ; Alzeidi, N. : User-Based Testing of GUI Sequences and Their Interactions. In: *Proceedings of the 12th International Symposium on Software Reliability Engineering*. Washington, DC, USA : IEEE Computer Society, 2001 (ISSRE '01). – ISBN 0–7695–1306–9, S. 54–

[123] Xie, Q. ; Memon, A. M.: Model-Based Testing of Community-Driven Open-Source GUI Applications. In: *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 0–7695–2354–4, S. 145–154

[124] Xie, Q. ; Memon, A. M.: Designing and comparing automated test oracles for GUI-based software applications. In: *ACM Trans. Softw. Eng. Methodol.* 16 (2007), Februar, Nr. 1, S. 4+. – ISSN 1049–331X

[125] Yatake, K. ; Aoki, T. : Automatic generation of model checking scripts based on environment modeling. In: *Proceedings of the 17th international SPIN conference on Model checking software*. Berlin, Heidelberg : Springer-Verlag, 2010 (SPIN'10). – ISBN 3–642–16163–4, 978–3–642–16163–6, S. 58–75

[126] Zhu, H. ; He, X. : A Theory of Testing High Level Petri Nets. In: *Proceedings of the International Conference on Software: Theory and Practice, 16th IFIP World Computer Congress*, 2000, S. 443–450

URLs

[URLa] Altia / Rhapsody connection. http://www-304.ibm.com/partnerworld/gsd/solutiondetails.do?solution=45525&expand=true&lc=en.

[URLb] CAMELEON project web site. http://giove.cnuce.cnr.it/projects/cameleon.html.

[URLc] Depth-first search. http://www.thi.informatik.uni-frankfurt.de/AlgorithmenTheorie/WS1011/rep/Tag%201/Kapitel3.pdf.

[URLd] Home page automotiveHMI. http://www.automotive-hmi.org/index.php?id=5.

[URLe] Home page Elektrobit. http://www.elektrobit.com/.

[URLf] Home page GUIDE. http://www.eb-guide-blog.com/hmi/eb-guide-studio/.

[URLg] Home page IRIS. http://www.sme.de/index.php/en/geschaeftsbereiche/automotive-infotainment/iris.

[URLh] Home page MODENA. http://www.berner-mattner.com/en/berner-mattner-home/products/modena/index.html.

[URLi] Home page VAPS XT. http://www.loyola.com/gis/partners/presagis/vaps-xt.html.

[URLj] Homepage StateChartXML . http://www.w3.org/TR/scxml/.