Generating Tests for Feature Interaction

Sebastian Benz

BMW Car IT

Technische Universität München

Institut für Informatik
der Technischen Universität München

Generating Tests for Feature Interaction

Sebastian Benz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Bernd Brügge, Ph.D.

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Ina Schieferdecker,
   Technische Universität Berlin

Die Dissertation wurde am 20.08.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09.04.2010 angenommen.

Abstract

Modern vehicles integrate an increasing number of different entertainment, information, telematics, and communication functions into their infotainment systems. The interaction between these features is often error prone and requires extensive testing in order to ensure correctness and reliability. The number of features of infotainment systems is expected to increase even more in future vehicle generations. As a consequence, the number of feature interactions increases as well. Hence, new approaches to testing are necessary in order to cope with the complexity of future infotainment systems. In this thesis we present a new approach to test case generation that enables the systematic coverage of feature interaction scenarios.

A prerequisite for test case generation is a test model that describes potential error prone paths and that limits the state space to a feasible size. In this thesis we introduce such a test model, which is based on task models. Task models describe the tasks the system is able to perform in interaction with its environment. Furthermore, they provide means to describe the temporal relations between tasks and thereby describe the space of all sequential task executions. We introduce a new task modeling language *TTask*. *TTask* provides a formal foundation that enables the systematic selection of such sequential task executions. Furthermore, we define new test selection criteria that enable the systematic coverage of "interesting" task executions, namely the ones that involve feature interactions.

A task model is, as any test model, an abstraction of the system under test (SUT). Hence, generated test cases are abstractions as well. The automated execution of these test cases requires additional information to bridge the abstraction gap between test case and the SUT. The transformation varies depending on different testing concerns, such as test goal, test setup and test phase. Therefore each testing concern requires a separate transformation of abstract test cases into executable test scripts. In this thesis we present a solution for test case instantiation that uses aspect-orientation as a means to encapsulate test concerns and hence reduces the test cases instantiation effort by reuse.

Kurzfassung

Die Anzahl der Funktionen in modernen Fahrzeugen wächst stetig. Unvorhergesehene Interaktionen zwischen Funktionen führen oft zu Fehlern. Um diese Fehler aufzudecken, sind systematische Tests notwendig. Diese Arbeit stellt einen neuen Ansatz zur Testfallgenerierung vor, der den systematischen Test von Interaktionen ermöglicht. Die Generierung von Testfällen erfordert ein geeignetes Testmodell, das potentielle Fehlerszenarien beschreibt und den Zustandsraum auf eine beherrschbare Größe einschränkt. Der vorgestellte Ansatz erreicht dies durch die Verwendung von Aufgabenmodellen. Diese Arbeit führt TTask, eine neue Modellierungssprache für Aufgabenmodelle, ein, sowie darauf aufbauende Testauswahlkriterien, die gezielt kritische Aufgabenabfolgen auswählen. Die so erreichte systematische Abdeckung von Funktionsinteraktionen wird anhand realer Systeme der BMW Group gezeigt.

Acknowledgements

First of all, I would like to thank Prof. Dr. Dr. h.c. Manfred Broy for supervising this thesis and for his always helpful feedback. Prof. Dr. Ina Schieferdecker I would like to thank for being the second reader and for her helpful and detailed comments.
I am grateful to my colleagues of BMW Car IT for providing a creative and productive environment that gave me the possibility to fully focus on my thesis. Special thanks go to Dr. Reinhard Stolle for his guidance during my work, his insightful comments and the many discussions that had a major influence on me. I also would like to thank Dr. Holger Grandy, Prof. Dr. Harald Heinecke, Maximilian Leinweber, Oliver Nölle, Oliver Scheickl, Dr. Kathrin Scheidemann and Dana Wong for reading and commenting on my thesis.
Furthermore, I would like to thank Markus Herrmannsdörfer, Elmar Jürgens and Sebastian Winter of the Department of Software & Systems Engineering for fruitful discussions and feedback.
I am grateful to my family, Julia, Jutta and Wolfgang Benz - knowing that I can always count on them makes many things easier. Finally, I would like to thank Manti Schwarzkopf for her encouragement, support and patience while I was working on my thesis.

Contents

Part II Test Case Generation

1 Introduction

Computers, once used only by the technologically savvy, have become integral to our everyday lives. With the advancement of computing power, the trend has moved towards integrated services. Modern mobile phones, for example, not only offer communication services, but also internet browsing, meeting organizers, camera functionality and entertainment features such as music and video. With the plethora of features on each device, this lends to the opportunity of additional features building its functionality on existing ones. For instance, the integration of a photo camera and internet access enables instant photo sharing with others.

This however comes at a cost. Developing software for these systems becomes more difficult due to the increased complexity. With each extra feature, the number of potential interactions between new and existing features increases and the potential for faults increases as more unforeseen situations can lead to unexpected side-effects. To counter this, new approaches are needed to cope with the increasing number of feature interactions new devises will have. This is especially true in the automotive industry where new vehicles have entertainment, information and communication functions integrated into the onboard computer.

These functions are combined in a so called automotive infotainment system (AIS). For example, the AIS of a current BMW vehicle, combines a multimedia player, navigation system, telephony, telematic services, and internet access. Figure 1.1 shows the infotainment system of the current BMW 7-Series with IDrive controller and display. The whole AIS integrates more than 1200 different features.

AIS belong to the class of multi-functional, distributed systems [DGP^{+}04]. Such a system consists of multiple features and its behavior is defined by the behavior of its features and by the interactions between its features. The situation when one feature modifies the behavior of another one is referred to as *feature interaction* [Zav03]. There are two forms of feature interaction: *intentional* and *unintentional feature interaction*. A system exhibits intentional feature interaction, when one feature influences the functional behavior of another feature in a specified and observable fashion. An example for an intentional feature interaction is, when the CD playback is interrupted by an incoming telephone call. An unintentional feature interaction takes place,

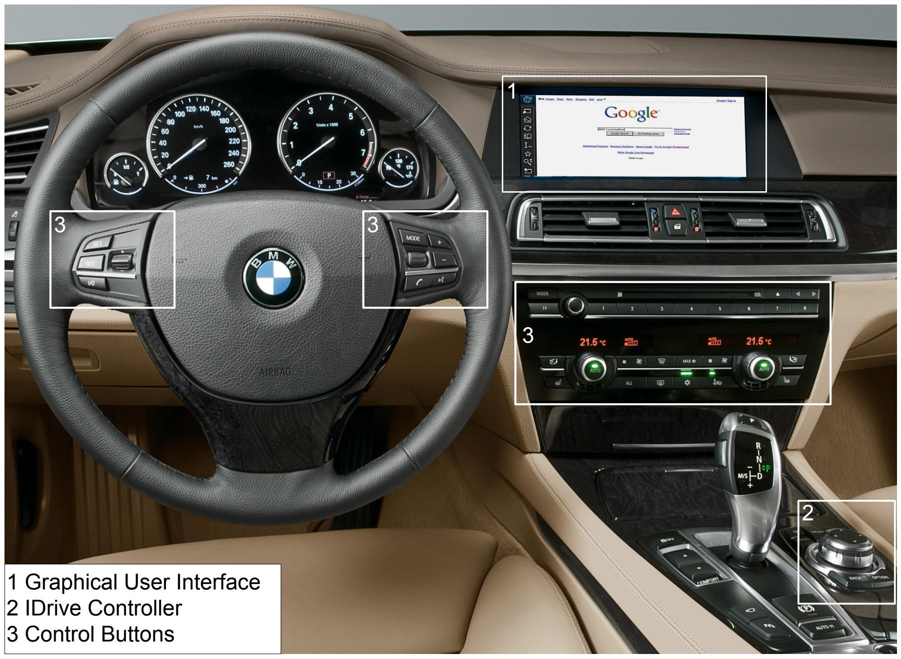


Fig. 1.1. Automotive Infotainment System.

when the execution of one feature influences the behavior of another feature in an unintended fashion. For example, when the user adjusts the seat, the bus load increases. When the increased bus load delays messages between graphical user interface (GUI) and navigation, an unintentional feature interaction between the feature seat adjustment and feature navigation takes place.

In practice, feature interactions are often error prone. In particular, scenarios where multiple intentional feature interactions interfere often lead to faults. For example, the media player may be suspended simultaneously by an incoming call and by a check control message and does not resume correctly. Unintentional interactions are often critical as well, because they are not specified and thus hard to predict. For example, the delayed bus messages might result in a race condition that causes the navigation advice not to appear.

Future generations of AIS are expected to include even more features than the existing generation due to the integration of new driver assistance systems and increasing connectivity. Furthermore, feature interactions are used to further enhance the capabilities of existing features. For instance, the active route in the navigation system can be used to reduce fuel consumption by adapting the brake-energy regeneration to the upcoming route profile. Therefore, systematically avoiding and detecting errors that result from feature interactions is an important prerequisite in order to cope with the complexity of future AIS. However, manually testing feature interactions in AIS is not feasible due to the large number of features and the resulting large number of possible feature interactions.

This thesis addresses the problem of generating executable test cases that cover critical feature interaction scenarios. In the first part of this thesis we introduce a modeling language that enables the specification of feature interactions and thereby enables the generation of test cases that explicitly cover feature interaction scenarios. Such a test model is always an abstraction of the real system [PP05]. Hence, generated test cases are abstractions as well. The automated execution of these test cases requires additional information to bridge the abstraction gap between test case and the SUT. This problem is called test case instantiation. In the second part of this thesis we address the problem of test case instantiation and present an approach that enables the generation of executable test scripts.

1.1 Problem Summary

This thesis has been created as part of the preparation for future generations of infotainment systems. In order to determine the main causes of faults in infotainment systems, we performed a study analyzing the faults that occurred during the development of an actual AIS. The study discovered that 40% of all severe faults involved feature interactions. This shows that by focusing on feature interaction scenarios, one can detect a large portion of faults.

Model-based test case generation enables the systematic test of specific system properties. For example, we used model-based testing at BMW Group to test infotainment systems. We generated test cases from the specification of the dialog behavior of GUIs. The generated test cases were successfully used for text testing, menu behavior tests and as test preambles for manual tests. Although our results indicate that model-based test case generation can be successfully applied in the automotive domain, we discovered that defects that result from feature interactions could not be found by test case generation from the dialog model. The problem was that feature interaction scenarios were not explicitly defined in the existing behavior models, such as the dialog model. The existing models focused on component behavior and on inter-component communication and did not explicitly describe feature interactions.

To understand why feature interactions are not modeled during development, we analyzed the current development process: when the development of an AIS starts, it is decomposed based on functional criteria into its components. These functional components are developed by different suppliers. Finally, the original equipment manufacturer (OEM) integrates them into the system. Features crosscut different software components and due to their crosscutting nature, features cannot be explicitly defined at software component level. However, different software components are developed by different suppliers. Therefore, it is the task of the OEM to test features and their interactions when integrating software components into the system. In order to systematically test features and their interactions, an appropriate test model is required.

One solution is to use a system model that describes the behavior of the whole system for test case generation. Such a system model implicitly describes feature interactions as well. Thereby, the generation of test cases that cover all paths in the system model results in the complete coverage of all feature interaction scenarios. However, in practice such an approach is not feasible: For example, a statechart that defines the dialog behavior of the GUI of an actual BMW vehicle, comprises more than 5,000 states, more than 1,500 variables, and more than 12,000 transitions. Thus the number of possible paths is enormous. A system model that comprises the dialog model, the telephone model, and the navigation model would increase the state space even more. Together with the fact that currently the automated execution of a test case can take up to several hours due to time-consuming test oracles, this leads to the conclusion that path coverage is not even remotely feasibly. Therefore, the number of generated test cases should be as small as possible and should focus on critical paths. However, feature interactions are not explicitly defined in the current development process. Hence, it is not possible to purposefully select critical paths that involve feature interactions.

In order to solve this problem a test model is required that has a manageable number of states but still describes feature interactions. Existing approaches use specification models to limit the number of tested paths for integration testing [PFH$^+$06, HIM00, FAM06]. They use scenario descriptions or use cases for test generation. However, scenarios or use cases focus only on specific features. In order to systematically cover feature interaction scenarios, we need a suitable model at a higher abstraction level without abstracting away feature interactions. More precisely, the model should describe intentional and unintentional feature interaction scenarios. Furthermore, the model should enable generation of test cases that systematically cover these feature interaction scenarios. The first problem for which a solution is proposed in this thesis is the following.

Feature interaction is often error prone. In order to systematically test feature interaction, a suitable test model is necessary that describes intentional and unintentional feature interaction scenarios and that enables an automated generation of test cases.

Test case generation faces an abstraction dilemma. On the one hand, test models must be abstractions that focus on critical system properties, on the other hand, generated test cases must be detailed enough to describe concrete test stimuli and system reactions: The dilemma is that test cases, which are generated from a test model, are abstractions as well. In order to execute such a test case, the abstraction gap between abstract test case and SUT must be bridged. Therefore, we differentiate between a test case and a test script. A test script is an executable test case that contains test stimuli and test oracles. The mapping of abstract test cases to executable test scripts is called *test case instantiation* and is an essential part of the model-based testing process. Test case instantiation is often time-consuming. In the case studies in [UL06], test case instantiation took about 25-45% of the modeling time. Embedded systems in particular require complex test setups to simulate the physical environment and to observe the behavior of the system. In the automotive infotainment domain, for example, the content of the graphical user interface is observed using screen grabbers, audio signals are observed by microphones, and haptic user inputs are simulated using special robots. Test case instantiation is complicated by the fact that it varies depending on a number of concerns: test goal, test setup and test phase. For instance, the same abstract test case must be transformed into different test scripts in order to be executable in different test setups that require different test stimuli and test oracles. We refer to a specific combination of test setup, test goal and test phase as a *test focus*. Thus, for each *test focus* a new test case instantiation must be defined. However, often testing concern specific test case instantiations crosscut different test focus definitions, such as a test oracle that is used in multiple test focuses. Hence, the second problem we address in this thesis is:

A test focus must be defined in such a way as to handle the complexity of test case instantiation and to enable easy test focus adaptation to different testing contexts.

The abstraction dilemma is especially relevant for feature interaction testing because the test model is at a high abstraction level. Therefore, an approach for feature interaction testing must solve both problems in order to be applicable in practice.

1.2 Solution Summary

The goal of this thesis is to enable the systematic test of feature interaction scenarios where errors are likely to occur. This thesis describes an integrated approach of model-based test case generation that focuses on feature interaction. The approach covers the generation of test cases and the subsequent test case refinement and instantiation in order to create executable test cases. Although the approach is introduced in the context of automotive infotainment system, it is not specific to the automotive domain. In fact, it can be applied to any multi-functional system where faults are often caused by feature interactions. Our approach is divided into three steps:

1. **Test Case Generation:** Test cases are generated from a test model at a suitable abstraction level, namely *tasks* and their *temporal dependencies*. Such a *task model* describes the interaction between user, environment and system in the form of tasks and their temporal dependencies. Tasks represent a view on a system that corresponds to the crosscutting nature of features. This thesis introduces a method of task modeling that enables the generation of test cases. Based on such a task model, we define *test selection criteria* that enable the generation of test cases that explicitly cover critical feature interaction scenarios.

2. **Test Case Refinement:** The resulting test cases are task sequences. A task model describes no input or output behavior. Thus, a task sequence contains no input or output behavior. In order to stimulate and monitor the system, task sequences must be enriched with missing behavior. Based on the assumption that software component-specific (test-)models contain input and output behavior, task models are mapped to these component models. Based on this mapping, task sequences can be enriched with the missing input and output behavior from the component models.

3. **Test Script Generation:** Test case instantiation transforms a test case into an executable test script. The instantiation depends on different test-

ing concerns, such as test goal, test setup and test phase. These concerns are encapsulated into aspects in order to decrease the complexity of test case instantiation and to enable the reuse in different testing contexts[Ben08].

The main contributions of this thesis are:

Task-based Test Case Generation:

- A task-based modeling approach to describe feature interaction for test case generation.
- New coverage criteria based on task models.

Task Sequence Refinement:

- A mapping approach from task models to existing software component models.
- A method that enriches task sequences with component-specific input and output behavior.

Test Script Generation:

- A taxonomy of test case instantiation concerns.
- A modular approach for test case instantiation based on aspects and a new language for it.

Implementation:

- The *TTask* modeling notation.
- Test case generator for *TTask*, based on the model checker SPIN.
- Refining of *TTask* sequences based on existing component models.
- *AspectT*: an implementation of an aspect-oriented test case instantiation language.

Case Study:

- A task model that models functionality of a current AIS.
- A mapping of the task model to existing component models.
- Implementation of different test instantiation concerns using *AspectT*.
- Generation of executable test scripts for different coverage criteria.
- Evaluation of the proposed test selection criteria.

1.3 Organization of this Thesis

This thesis comprises four parts. The first part (Chapters 2 and 3) describes the fundamentals of test automation and gives an introduction into the automotive infotainment domain. In the second part (Chapters 4 and 5), we describe our approach of task-based test case generation. The third part (Chapters 6 and 7) focuses on the instantiation of abstract test cases created from task models. In the fourth part (Chapters 8-10), the introduced concepts are evaluated in the context of an actual BMW vehicle. The discussion of related work to this thesis is not covered in one separate chapter. Instead, it is distributed across the different parts of this thesis due to the diversity of the covered topics. Figure 1.2 gives an overview over the outline of this thesis.

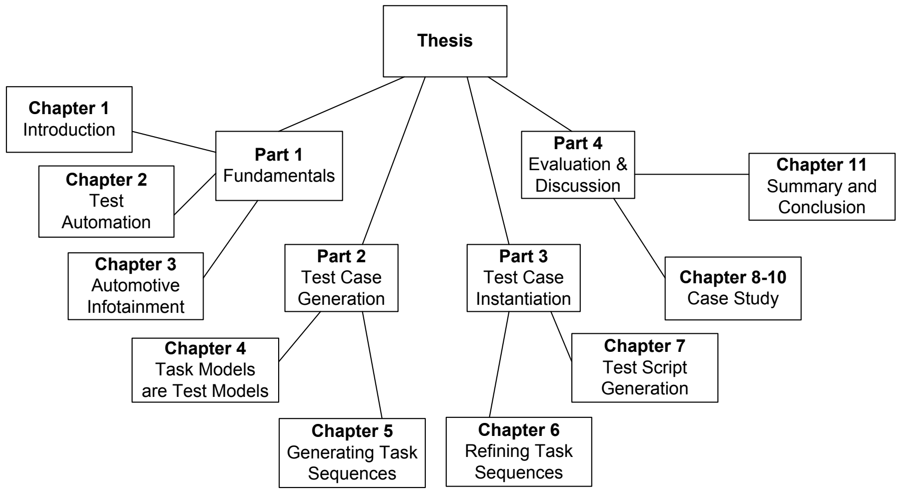

Thesis
Chapter 1 Introduction
Part 1 Fundamentals
Chapter 2 Test Automation
Chapter 3 Automotive Infotainment
Part 2 Test Case Generation
Part 3 Test Case Instantiation
Part 4 Evaluation & Discussion
Chapter 11 Summary and Conclusion
Chapter 8-10 Case Study
Chapter 4 Task Models are Test Models
Chapter 7 Test Script Generation
Chapter 5 Generating Task Sequences
Chapter 6 Refining Task Sequences

Fig. 1.2. Outline of this thesis.

Chapter 2. Test Automation. The topic of the second chapter is test automation with a special focus on test case generation and test case instantiation. We illustrate the concepts of test case generation using an example from the automotive infotainment domain. Based on the example we describe different techniques of behavior modeling and test case generation.

Chapter 3. Automotive Infotainment Systems. The third chapter gives an overview of the architecture and the development process of automotive infotainment systems. Based on the architecture, we introduce a classification of typical error scenarios and the results of an analysis of errors which occurred during production development of a BMW infotainment system. Furthermore, we outline the current testing process and the challenges during component and integration testing.

Chapter 4. Task Models are Test Models. The fourth chapter introduces task models. Conventionally, they are used in the early phase of model-based user interface (UI) development to describe the possible tasks the user can perform in interaction with the system. First we argue that task models describe critical feature interaction scenarios and are therefore a suitable abstraction for test case generation. Then we compare different task modeling notations with respect to potential test case generation. We conclude that existing task modeling notations are insufficient for test case generation. The rest of Chapter 4 introduces *TTask*, a new task modeling notation that meets the needs of test case generation.

Chapter 5. Generating Task Sequences. This chapter describes test case generation based on *TTask* and the model checker SPIN [Hol97]. A *TTask* model describes the space of all possible task sequences where each sequence is a potential test case. We define test selection criteria that cover critical task sequences. This chapter describes the transformation of a *TTask* model into a *Promela* [Hol97] model and the implementation of the test selection criteria in temporal logic.

Chapter 6. Refining Task Sequences. Chapter 6 describes the refinement of task sequences using component models. A task sequence must be refined in order to reduce the abstraction gap between the task sequence and the SUT. We describe the mapping between a task model and existing component test models. Based on this mapping we describe the enrichment of task sequence with additional input and output behavior from component models.

Chapter 7. Test Script Generation Using *AspectT*. This chapter presents the last step in test case instantiation: the test script generation. The test case instantiation varies depending on different testing concerns. We present *AspectT*, an aspect-oriented language for test script generation that provides means to handle these crosscutting testing concerns in an efficient way.

Chapter 8. From Use Cases to Task Models This chapter demonstrates, how to create a *TTask* model from a use case-based specification. The study is performed for the software update functionality of a current BMW vehicle.

Chapter 9. Case Study. This chapter presents a case study from an actual BMW vehicle. The case study demonstrates the application of task-based test case generation and test case instantiation for a real world example. In the case study, a *TTask* model of multiple features of the infotainment system is described. From the *TTask* model test cases are generated and instantiated using message sequence charts (MSCs) and statecharts.

Chapter 10. Test Selection Criteria Evaluation. This chapter presents the results of our evaluation of the introduced test selection criteria. The criteria are evaluated in the form of an empirical analysis of the faults that occurred during the development of a BMW infotainment system.

Chapter 11. Summary and Conclusions. The last chapter summarizes this thesis and draws conclusions.

Part I

Fundamentals

2

Test Automation

Test automation is the automation of test case generation and test case execution. The main goal of test automation is to enable a systematic testing process that continuously assesses the correctness and the quality of a system. This chapter introduces the principles, activities, and goals of test automation and is structured as follows. The first section introduces the general concept of testing. The second section introduces model-based test case generation. The third section introduces test case instantiation, the transformation of generated test cases into executable test scripts.

2.1 Testing

Testing is defined in *IEEE Software engineering body of knowledge* [IEE04] as:

Testing is a technique for evaluating product quality and also for indirectly improving it, by identifying defects and problems. Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior.

An important role of testing is to evaluate and assure the quality of a system by examining whether it meets its requirements and expectations. This is accomplished by executing the SUT with specific inputs in order to find failures in its behavior. A *failure* is the inability of the system to perform a required function. It is caused by a *fault*, which is a condition that causes a system to fail in performing its required function. The manifestation of a *fault* in a system is an *error*: a discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

Testing can be differentiated based on the information that is used for test design. For instance, this can be seen comparing black box testing to white box testing. In *black-box testing*, the tests are created from requirement documents. During test design no information about the internal structure of the system is used, the SUT is treated as a black box. On the other hand, in *white-box testing*, the implementation of the system is used for test case design. An example for *white-box testing* is when tests are designed in order to cover each statement in a procedure.

2.2 Model-based Testing

Testing is always based on an abstract model of the system [PP05]. Even manual testing is model-based, because every tester has a mental model of the system to create test cases from. There are four main approaches of model-based testing [UL06]:

1. **Generation of test input data from a domain model.** The test model describes the domain of the test input data. A test generator selects and combines subsets of those values to produce test inputs.
2. **Generation of test cases from an environment model.** Here, the test model describes the expected environment of the SUT. The test model generates sequences of calls to the SUT. However, the generated test cases do not specify the expected outputs of the SUT.
3. **Generation of test cases with oracles from a behavior model.** This approach generates test cases that include oracle information, such as the expected output values of the SUT. Hence, the test model must describe the expected behavior of the system.
4. **Generation of test scripts from abstract test cases.** In this approach, an abstract test case is given, for example, in the form of sequence diagrams [ITU96, OMG03]. This test case is then transformed into an executable test script that includes the corresponding system calls and test oracles which are necessary for an automated execution.

In this thesis we focus on the last two approaches of model-based testing. In the remainder of this chapter, we introduce these approaches in more detail.

2.3 Model-based Test Case Generation

We focus on the automated generation of black-box tests from a formal behavior model of the system. Such a behavior model describes the interaction between the system and its environment omitting details about the actual implementation of the system. Each execution path in the behavior model is a potential test case that tests the specified execution path in the SUT. A behavior model usually describes an infinite set of execution paths. However, only a finite subset of these paths can be executed in the form of test cases. A *test selection criterion* restricts the space of all test cases to a finite set by defining a property that must be fulfilled by each test case. *Test case generation* is the automation of test case selection using a *test case generator*. A *test case generator* automatically selects all test cases from a behavior model that fulfill a given test selection criterion. We refer to a behavior model that is used for test case generation as a *test model*. The selected test cases are *abstract test cases* that are on the same abstraction level as the test model. Figure 2.1 depicts the general idea of model-based test case generation.

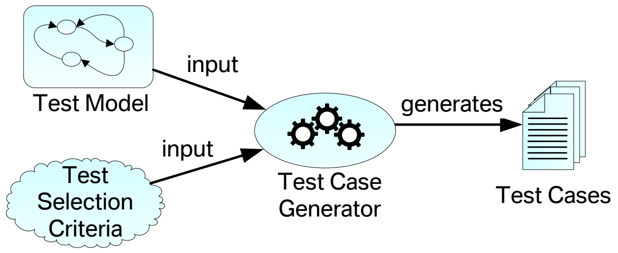


Fig. 2.1. Model-based test case generation.

2.3.1 Test Model

Model-based test case generation requires an appropriate model that describes the properties of a system or its environment that should be tested. This can either be a dedicated model for test case generation or a model that is part of the specification. A test model must be at a suitable abstraction level: it must describe critical behavior and avoid unnecessary detail.

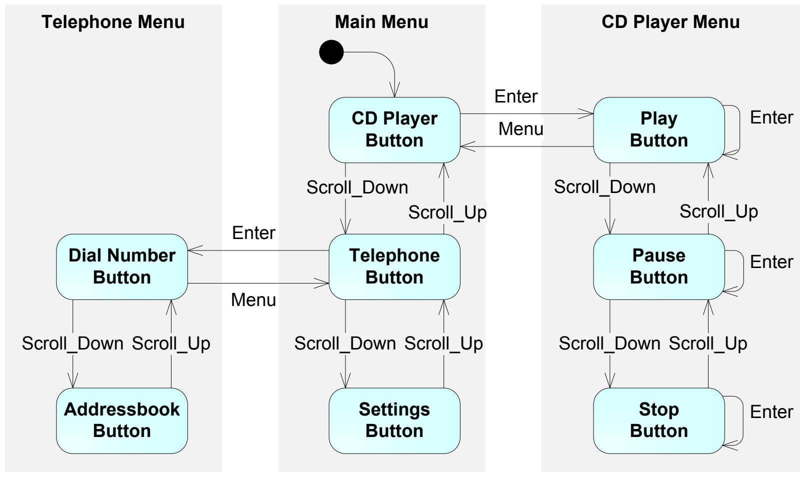


Fig. 2.2. Example test model for an automotive GUI.

Figure 2.2 shows such a test model: a state machine that describes the focus behavior of an automotive GUI by the possible cursor changes between its buttons. The test model abstracts from the actual implementation of the GUI by focusing only on its visible behavior. Thereby, the model describes the space of all possible focus changes, which is potentially infinite. However, only a finite number of test cases can be executed. Therefore, the space of all possible focus changes must be reduced to a finite set. The process of selecting specific sequences for testing is called test case generation and is performed by a test case generator. Test cases are selected by a so called *test selection criterion.*

2.3.2 Test Selection Criteria

For a given program and its specification a test selection criterion specifies the properties that must be satisfied by a set of test cases [Jal91]. Test case generation based on test selection criteria is called *coverage-based testing.* However, the intention is not to guarantee test coverage of the SUT - its structure may be quite different from that of the model. Rather, they represent the aspects of the system behavior that the engineer wants to test [UL06].

There are different coverage-based test selection criteria.

- **Structural Coverage Criteria** cover certain structural parts of the test model. These properties often depend on the semantics of the used modeling language. In the state machine example in Figure 2.2, typical coverage criteria are *state coverage* or *transition coverage.* An overview on structural test selection criteria can be found in [UL06, OXL99].
- **Fault-based Coverage Criteria.** Fault-based testing demonstrates the absence of a predefined set of faults. Hence, *fault-based coverage criteria* systematically select test cases for a given fault class. For instance, a behavior model is systematically mutated and test cases are generated that distinguish each mutant from the original model [TSL04].
- **Domain-specific Coverage Criteria** cover properties that are specific to the respective domain. For example, the GUI of an AIS should save the current state when the car is switched off, until it is switched on again. Hence, a domain-specific coverage criterion covers all situations where the GUI should save its current state.
- **Resource-based Criteria** cover the SUT based on the available test resources. For example, test cases are generated randomly and are executed for a give amount of time.

Another class of test selection criteria is the *explicit test case selection.* Here, one test case is selected that tests one specific property in the behavior model. This can be used as an addition to coverage-based test case selection in order to test properties that are not covered by the coverage criteria. Furthermore, this can be used to generate *test preambles.* A *test preamble* establishes a specific state in the SUT which is a precondition to perform other tests. For example, a tester writes a test case that tests the incoming call screen and generates a test preamble that triggers the system to show the screen.

A rather trivial test selection criterion is the random selection of test cases. However, in practice this has proven a valuable extension to classical coverage criteria [CLOM07, DN84]. Furthermore, random selection of test cases can be used to test the SUT for a longer period of time.

The actual selection of test cases is performed by a test case generator. The test case generator is responsible for selecting all test cases from a given test model that fulfill a given test selection criterion. The actual implementation of the test case generator depends on the underlying modeling language.

2.4 Test Case Instantiation

Model-based test case generation produces a potentially large number of test cases. The generated test cases are abstract test cases that are at the same abstraction level as the test model. It is desirable to automate the execution of these test cases to reduce manual test execution effort and to enable a continuous testing process. For instance, a test case generated from a state machine model consists of events that trigger certain transitions and states. Events represent test inputs and states represent the expected test outputs for these inputs. For testing a concrete system, such as an embedded automotive control unit, the abstract events must be mapped to the corresponding system inputs that stimulate the SUT. For instance, the event "Incoming Call" must be mapped to the bus message that signals an incoming call. Only then this event is executable in an automated setting. Likewise, a state has to be mapped to an observable system property that represents this state. The component that observes the system and decides if it behaves correctly is termed the *test oracle*.

The mapping of abstract test cases to executable test scripts is called *test case instantiation* [PERH04] and is either performed by a translator, which adds the missing information (e.g., mapping of equivalence classes to concrete data to overcome data abstraction), or by an adapter that encapsulates the missing information. Both test case instantiation approaches use driver components. Such a driver component encapsulates the access to a device that is used to stimulate or monitor the system, for example, a component that encapsulates the access to the screen grabber.

Adaptation-based instantiation approaches encapsulate the missing information to bridge the abstraction gap between test case and SUT into a specific adapter. Such an adapter interprets a test case and executes during interpretation the corresponding test triggers and test oracles. Figure 2.3 shows the adapter-based test case instantiation approach. For example, the testing and test control notation TTCN-3, a test specification and test implementation language, integrates an adaptation-based instantiation approach [GHR+03a]. In TTCN-3, each abstract test input is mapped to an adapter that triggers the system under test and each system output is adapted to its corresponding abstract output.

Transformation approaches translate abstract test cases into executable test cases and are typically based on model-to-text transformation frameworks. During the translation process the test cases are enriched with the missing information to bridge the abstraction gap between test model and SUT. For instance, an abstract user input is translated into a call to a driver component that simulates the user input by a corresponding bus message. Figure 2.4 shows the translational test case instantiation approach. The transformation

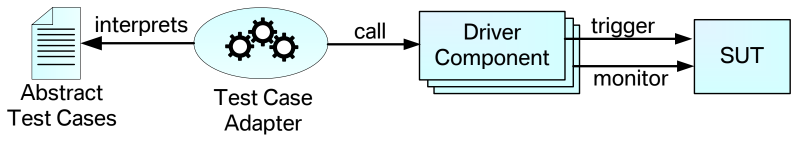


Fig. 2.3. Adapter-based test case instantiation.

is performed based on *mapping rules* that contain the information on how to translate an abstract element in the test case. These mapping rules may be code generation templates, for example.

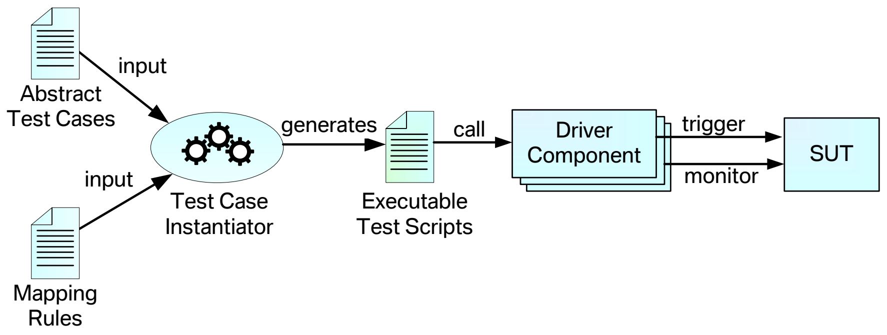


Fig. 2.4. Translational test case instantiation approach.

2.5 Automated Test Case Execution

Test case generation and test case execution are two essential parts of model-based testing. They can be combined in either an *online* or an *offline* fashion, depending on when the test cases are executed during test case generation. In *online testing*, test cases are executed while they are generated. For example, when test cases are executed on-the-fly during random test case generation. This is useful when the SUT is tested for a longer period of time, for example, during overnight testing. Adaption approaches of test case instantiation are especially suitable for online test case execution, because test cases can be interpreted during creation.

In *offline testing*, test cases are executed after they are generated. This enables the storage of generated test cases in a test management system in order to use them for regression testing. Translational test case instantiation approaches are well suited for offline testing, because they transform abstract test cases into an intermediate format that can be stored in a test management system and that can be executed separately. Furthermore, they enable further methods of test suite reduction.

In Chapter 7 we introduce an approach for test case instantiation that enables the combination of both online testing and offline testing.

2.6 Summary

This chapter gave a short introduction into model-based test case generation. In this thesis we focus on the generation of test inputs and test oracles from a behavior model of the system. Test cases are generated based on test selection criteria that restrict the space of all possible test cases. There are a large number of different selection criteria, the appropriate one is dependent on the modeling language, the targeted system and the system's domain. However, test case generation is only one part of model-based testing. Another important role is test case instantiation. A test model is an abstraction of the SUT. Hence, generated test cases are abstractions as well. Test case instantiation is the process of bridging the abstraction gap between abstract test cases and SUT in order to enable their automated execution. To conclude, effective model-based testing requires an appropriate modeling language that is able to express critical behavioral aspects of the system. Furthermore, test selection criteria must be available that enable the systematic coverage of these critical scenarios.

3

Automotive Infotainment Systems

This chapter introduces automotive infotainment systems (AIS) as examples of multi-functional interactive systems. AIS integrate a large number of different features. The interaction between these features is often error prone and requires extensive testing in order to ensure the correctness and reliability of an AIS. The number of features in an AIS is expected to increase even more in future vehicle generations. As a consequence, the number of feature interactions increase as well. New approaches for testing are necessary to cope with the complexity of future infotainment systems. In this thesis we present a new approach for test case generation that enables the systematic coverage of feature interaction scenarios. Our approach focuses on multi-functional interactive systems in general and is not limited to the automotive domain. However, in order to exemplify our approach we introduce it based on the example of AISs. Therefore, this chapter introduces the characteristics of AIS and shows why the systematic test of feature interactions is necessary.

This chapter is structured as follows. The first section introduces the architecture of AISs. The second section introduces feature interactions and shows their importance by a study of faults that occurred during the development of an actual BMW vehicle.

3.1 Architecture

AIS are interactive systems that are characterized by a strong interaction between system and environment. They integrate different features for communication, navigation, entertainment, and telematic services. The UI plays an important role in AIS. It must enable the safe usage of the AIS in different driving contexts. Therefore, AIS are multi modal systems that enable the interaction between user and system via different input and output modalities, namely visual, audio-based, and haptic modalities. This enables the user to interact safely with the AIS using the modality that is appropriate for the current driving context. However, the multi modality increases the complexity of AIS because all modalities must be synchronized in order to consistently present the system state to the user.

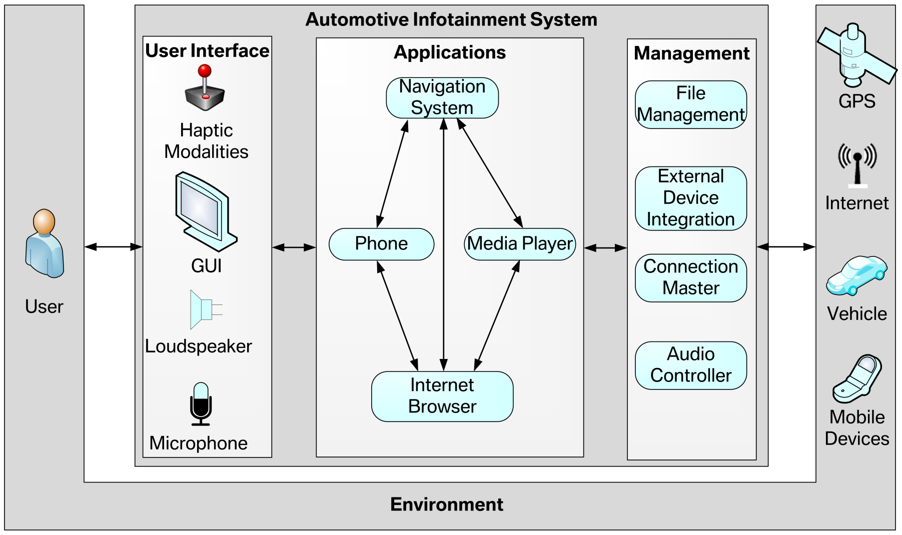


Fig. 3.1. AIS features and their interactions.

An AIS is also connected to other vehicle functions, such as the engine control or driver assistance systems. In particular the integration of driver assistance systems, such as the park distance control, leads to high demands on the reliability of an AIS. Another important characteristic of AIS is that it enables the integration of external devices such as mobile phone or media players in order to make these additional functionalities available in the driving context.

Figure 3.1 shows the three main building parts of an infotainment system and their interaction with their environment. They are in detail:

- The *user interface (UI)* of an AIS integrates the different input and output modalities. For example, the UI if a 7-Series BMW, integrates the IDrive controller, a speech interface, and different haptic buttons. Furthermore, it integrates the output modalities: graphical user interface (GUI) and audio-speakers.
- *Applications* implement the main feature of an AIS.
- *Management* features manage resources, external devices and other basic system services.

Each of these parts is composed of multiple software components that are distributed across different electronic control units. They interact via one or more communication buses. Figure 3.2 shows an example infotainment architecture. The infotainment ECUs communicate with each other via the media oriented systems transport (MOST) bus [MOS] and the head unit additionally communicates with other vehicle functions via the controller area network (CAN) bus [BOS91].

To conclude, AIS interact strongly with their environment. Furthermore, the different software components inside an AIS strongly interact with each other in order to implement the different features of an AIS.

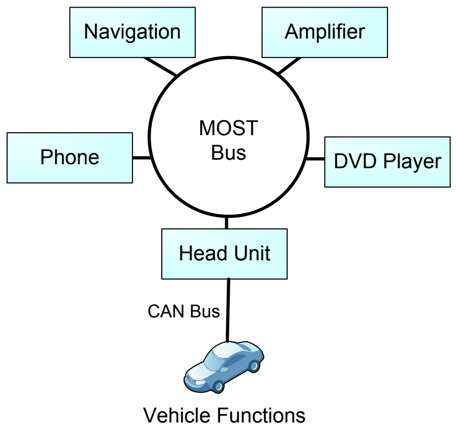


Fig. 3.2. Infotainment system architecture.

3.2 Feature Interactions in Infotainment Systems

From the user's point of view, an AIS is composed of different features, where a feature enables the user to perform a certain task. Features have observable behavior and can be triggered by the environment[EKS03]. For example, the feature Incoming Call Handling is triggered by an incoming call and the observable behavior is the notification popup in the GUI and the ring tone. Hence, when the system is regarded as a black-box, a feature comprises of the interactions between system and environment that are required to perform the corresponding task. Figure 3.3 shows the interactions between system and environment for the feature Incoming Call. The terms *feature* and *service* are often used synonymously. According to Broy et al. [BKM07] a service is a crosscutting interaction aspect of complex software systems, factoring out collaboration among software components required for fulfilling a certain task. Hence, both service and feature describe an interaction between environment and system. In this thesis we use the term *feature* instead of *service* because the problem of error prone interactions between different features (services) is commonly known as the *feature interaction problem* [BGK00, GL93].

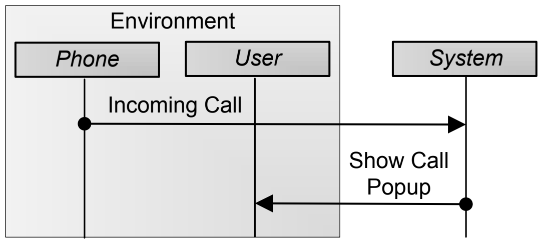


Fig. 3.3. Feature Incoming Call Handling.

The situation when one feature influences the behavior of another feature is known as *feature interaction*. The term feature interaction has first been described for telecommunication systems. Feature interaction in telecommunications is the expansion of existing telephone services (called features) with a new supplementary service that can change the behavior of pre-existing ones, alter them, or even crash the system. The phenomena are known as the “feature interaction problem” in the telecommunication industry [BGK00, GL93]. However, not all feature interactions are errors and in order to separate more clearly between erroneous feature interactions and correct feature interactions, we refer to erroneous feature interactions as *feature interference*.

We differentiate between two different forms of feature interaction: *intentional-* and *unintentional feature interaction*. Intentional feature interaction takes place when one feature influences the functional behavior of another feature in a specified fashion. An example for an intentional feature interaction is when the incoming telephone call interrupts the radio. The muting of the radio is the observable result of the specified feature interaction between radio and telephone. If the radio does not resume correctly after the incoming call is finished, the interaction between radio and telephone exhibits *feature interference*.

Unintentional feature interaction takes place when one feature influences another feature in an unspecified fashion that does not depend on a functional dependency. For example, when the user adjusts his seat, the bus load increases. When the increased bus load delays messages between GUI and navigation, an unintentional feature interaction between the feature seat adjustment and the feature navigation takes place. An unintentional feature interaction is not necessarily a fault. For example, when the delayed bus messages cause a delay of 100ms of the navigation advice, it is an acceptable unintentional feature interaction (as long as it is not explicitly specified otherwise). When the delayed bus messages result in a race condition which causes the navigation advice not to appear, the unintentional feature interaction exhibits a *feature interference*.

The distinction between intentional and unintentional feature interactions has also been introduced by different authors, however there is no consistent terminology across the literature. Zave describes intentional and unintentional feature interactions as *wanted* and *unwanted feature interactions*[Zav01]. Other authors speak of *intended* and *unintended* feature interactions [WEL07]. Weiss et al. use the terms *functional* and *non-functional* feature interactions following the concepts of functional and non-functional requirements [WE04, WEL07]. However, we use the terms *intentional* and *unintentional feature interaction* because they emphasize the contrast between intentional functional interaction and unintentional interactions that result from other non-functional properties.

In practice, feature interactions are often error prone. During the development of an actual infotainment system, a large fraction of the faults resulted from feature interactions. Figure 3.4 shows a statistic that describes the reasons for severe faults that occurred during development. The results show that 40% of all severe faults occurred in feature interaction scenarios. The study has been performed as part of a diploma thesis at BMW[Wol08]. During the study, the AIS was divided into its different features. All faults that result from the interaction between different features have been classified as feature interaction faults.

These faults were further analyzed by deciding whether they resulted from intentional and unintentional feature interactions. Figure 3.5 shows the results

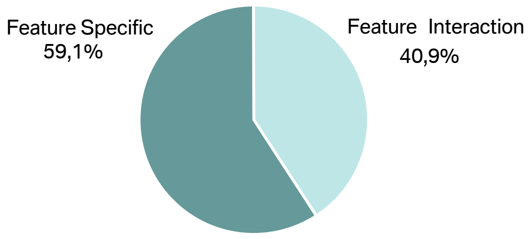


Fig. 3.4. Faults resulting from feature interaction.

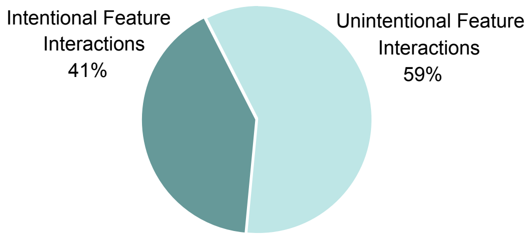


Fig. 3.5. Intentional versus unintentional interactions.

of this second study. The results show that most feature interferences result from unintentional feature interactions. This leads to the conclusion that a systematic approach is required for testing intentional as well as unintentional feature interactions.

The literature [KK98, Vel93, CGL$^+$94] discusses several reasons for feature interferences:

- *Conflicting goals:* features with the same preconditions but incompatible goals are in conflict.
- *Competition for resources:* features compete with each other for limited resources, which need to be partitioned among the features.
- *Changing assumptions on features:* features make implicit assumptions about their operation, which can become invalid when new features are added.
- *Design evolution:* features need to be added to meet new customer needs, and the system will need to interoperate with other vendors' systems.
- *Timing and race conditions:* Timing (i.e., at what time a particular event occurs, or for how long an event lasts) is always critical in distributed systems.

In AIS, feature interactions are often critical due to the current development process in the automotive domain. The functional decomposition of a system does not necessarily correspond to the features of a system. An AIS is decomposed into different software components based on functional criteria, in order to encapsulate specific functionalities. This enables the distributed development of an AIS where different suppliers develop different software

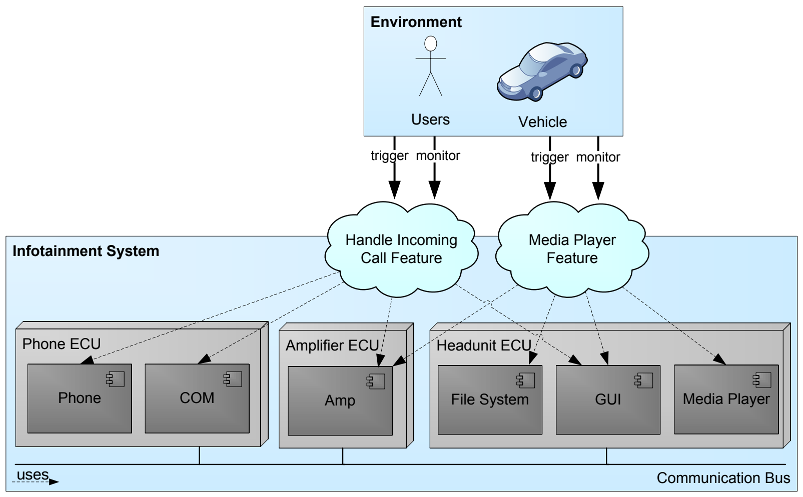


Fig. 3.6. Crosscutting nature of features.

components. However, features crosscut different software components. For example, in an AIS, most features use multiple software components: Figure 3.6 shows the software components that are responsible for the Handle Incoming Call Feature and the Media Player Feature. The Incoming Call Handling Feature crosscuts the Phone component, Communication component, AMP component and the GUI component. The Media Player Feature crosscuts the AMP component, File System component, GIO component and the Media Player component. Both features in Figure 3.6 use the components GUI and AMP. Feature interactions take place when both features use these components at the same time. However, the scenarios in which these interactions might occur are often not explicitly defined, because they cannot be described at software component level. Hence, when a supplier implements a software component, not all possible scenarios in which such a component is used, are known.

3.3 Summary

This section introduced automotive infotainment systems. AIS are distributed systems that have a strong interaction with their environment. Features crosscut different software and hardware components. Software components are developed by different suppliers. Thus during development, a system is decomposed into its software components which are specified and implemented separately. Hence, features and their interactions are often not explicitly specified. However, the study we presented in this chapter showed that feature interactions are often error prone. In order to cope with the increasing number of features in an AIS, feature interactions must be rigorously tested. In the following chapters we introduce our approach towards a systematic test of feature interactions.

Part II

Test Case Generation

Task Models are Test Models

This chapter proposes task models as a language to describe feature interactions. They describe the tasks that the user can accomplish by interacting with the system and their temporal dependencies. They were originally developed for formally describing the capabilities of UIs[DS03], but here they are used to model features and their dependencies. Based on the task dependencies, a task model implicitly spans the space of possible feature interactions. In this space of possible feature interactions we can then generate useful test cases.

The contributions of this chapter are:

- Introduction of task models as a means for describing feature interactions.
- Description of the concepts of the new task modeling language *TTask*.
- The definition of the semantics of *TTask*.
- Examples that demonstrate how to model feature interactions with *TTask*.

This chapter is structured as follows. In the first part, we reason that task models describe critical feature interactions and hence are appropriate for test case generation. Then, we evaluate different task modeling notations with respect to their suitability for test case generation. We conclude that existing task modeling notations are insufficient for test case generation. In the remainder of this chapter we introduce *TTask*, a new task modeling notation that meets the requirements of test case generation.

4.1 Task Models as Means to model Feature Interactions

This section introduces task models as means to model feature interactions using a system model that is based on action traces. Actions are basic activities in a system such as opening a popup screen or pressing a button. Actions are thought to be atomic and instantaneous [BDD$^+$92]. Based on a possibly infinite set of actions Act, the set of streams of actions, is denoted by:

$$Act^{\omega} = Act^{*} \cup Act^{\infty}$$

where Act^* denotes the finite streams (including the empty stream) and Act^∞ the infinite ones [BKM07, BS01]. We will use the following basic operations and relations concerning streams (following Broy et al. [BDD+92]):

- $s \circ t$ denotes the concatenation of t to the end of s. If s is infinite, then $s \circ t$ just yields s.
- $s \sqcup t$ denotes that s is an infix of t which is formally expressed by

$$\exists u, w.\, u \circ s \circ w = t$$

One run of a system is a trace which is a stream of actions. The behavior of a system S can be specified by all possible system traces:

$$S \subseteq Act^\omega$$

Each trace $s \in S$ is a potential test case. The actions that are performed by the environment are test inputs and the actions that are performed by the system are test outputs. In an ideal world, every possible trace $s \in S$ would be covered during testing. However, due to the potentially infinite behavior of a system, this is not feasible. Hence, testing should focus on the traces that are most likely to exhibit faults in the system. The previous chapter introduced the results of a study showing that feature interaction scenarios are often error prone. Our goal is to systematically cover these scenarios by test case generation. Hence, we want to select and test traces $s \in S$ that involve a feature interaction and therefore are likely to exhibit faults.

In order to obtain all traces that involve a feature interaction, feature interactions must be defined explicitly. Our approach is to describe features and their interactions in the form of a task model. A task model describes tasks and their temporal dependencies. A task is a goal driven interaction between system and environment, where a goal is a desired modification of the state of a system or a query to it [PMM97]. We define a task by a finite sequence of one or more actions:

$$t = \langle a_1, ..., a_m \rangle : a_i \in Act^*$$

For example, the task Start CD Playback has the goal to start the playback of a CD, which is defined by the action sequence: Press Play Button, Start Track 1. However, the notion of a task is always bound to a certain abstraction level. For example, the task Start CD Playback involves the action Press Play Button. On a lower abstraction level, this action itself might be declared a task, namely the task Press Button. As a consequence, a task model itself is always bound to a certain abstraction level.

A task t is performed in a trace $s \in S$ iff:

$$t \sqcup s$$

Therefore, a task model T models the behavior of a system S iff:

$$\forall t \in T.\, \exists s \in S.\, t \sqcup s$$

We can separate a system trace $s \in S$ that involves the execution of a task t into three sub streams s_0, s_1, t:

$$s = s_0 \circ t \circ s_1$$

The trace s_0 comprises all actions that are performed before the task t is performed. These actions may involve the execution of other tasks or additional interactions that are necessary to reach the goal. For example, in order to perform the task Start CD Playback, the user must first navigate to the play button. The necessary navigation steps depend on the previously executed tasks, for example, whether the user previously made a telephone call or listened to the radio. Therefore we can separate s_0 into two parts:

$$s_0 = h \circ p$$

Where h comprises the history of previously executed tasks and p comprises the actions that were necessary to "prepare" the system for the next task t. This differentiation corresponds to the situation during testing, when the tester wants to test sequences of task execution. For example, in the first part of a test the task Select Radio Station is tested and in the second part the task Start CD Playback is tested. Therefore, in order to test the task Start CD Playback, the tester must prepare the system by navigating to the Play CD Button from the radio menu which is the postcondition of the previously executed task.

Our goal is to model feature interactions. Therefore, we use a task model to model each feature by the tasks that can be accomplished using the feature. For example, the feature Incoming Call is modeled by the task Handle Incoming Call. Furthermore, a task model provides means to define the temporal dependencies between tasks. An example for such a temporal dependency is, when the execution of a task disables the execution of another task. The basic idea is to use these dependencies to model feature interactions.

For example, given a simple task model T that comprises of three tasks $a, b, c \in T$. The task a has a *disable* dependency to the task b

$$disable(a, b)$$

and the task c has an *enable* dependency to the task b

$$enable(c, b)$$

The temporal dependency *disable* between the tasks a and b defines that there is no trace in which the tasks a and b are performed sequentially, as long as the task c is not performed:

$$\forall s.\ \exists s_0, s_1, s_2.\ (s = s_0 \circ a \circ s_1 \circ b \circ s_2) \Rightarrow c \sqcup s_1$$

A feature might provide different interaction scenarios. For example, the interaction between environment and system of the feature Incoming Call varies depending on whether the user decides to accept or reject the incoming call. Figure 4.1 shows a sequence diagram for the Incoming Call that describes the Incoming Call feature. To model features with different interaction scenarios, a task can be decomposed into subtasks that describe corresponding sub goals. For example, the feature in Figure 4.1 is described by the task Handle Incoming Call. The different interaction scenarios of Incoming Call Handling are described by the four subtasks of Handle Incoming Call and their dependencies:

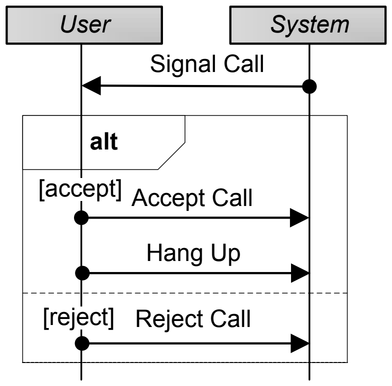


Fig. 4.1. Incoming Call feature.

1. **Signal Call:** which is performed by the system by showing the incoming call popup and by giving an audio signal.
2. **Accept Call:** which is performed by the user by selecting and activating the accept call button.
3. **Hang Up:** which is performed by the user by selecting and activating the end call button.
4. **Reject Call:** which is performed by the user by selecting and activating the reject call button.

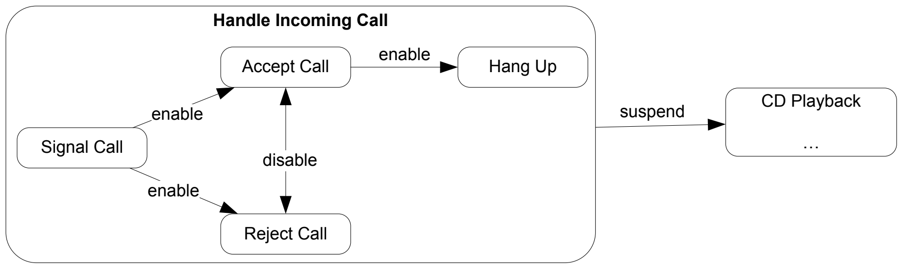


Fig. 4.2. The tasks for the feature Incoming Call.

Figure 4.2 shows a complete task model for the features Incoming Call and CD Player. The task model uses *enable* and *disable* dependencies to describe whether the execution of a task enables or disables the execution of another task respectively. Thus, when Signal Call has been executed, the tasks Accept Call and Reject Call are enabled. When one of these two tasks is performed, the other one is disabled. Thus the task model specifies that the task Handle Incoming Call is performed in all traces $s = s_0 \circ s_1 \circ s_2 \in T$ that hold:

$$\begin{aligned}&\text{'Incoming Call'} \sqcup s \Leftrightarrow \text{'Signal Call'} \sqcup s_0 \wedge \\ &\quad (\text{'Accept Call'} \sqcup s_1 \wedge \text{'Hang Up'} \sqcup s_2) \vee (\text{'Reject Call'} \sqcup (s_1 \circ s_2))\end{aligned}$$

A composite task is performed by executing its subtasks. The temporal dependencies between the subtasks define their valid execution orders. Figure 4.2

also shows a dependency called *suspend* between the tasks Handle Incoming Call and CD Playback. This dependency models an intended feature interaction between the features that are represented by these two tasks, namely that the CD playback is suspended when an incoming call is handled. The traces that are specified by this dependency are the ones we want to test in order to cover intentional feature interactions.

Furthermore, the execution of tasks can be interleaved with the execution of other, independent, tasks. For example, in a trace $s = s_0 \circ s_1 \circ s_2$ the task 'Move Seat' might be performed between the tasks Signal Incoming Call and Reject Call:

$$\text{'Signal Call'} \sqcup s_0 \wedge \text{'Move Seat'} \sqcup s_1 \wedge \text{'Reject Call'} \sqcup s_2$$

The interleaved execution of tasks is potentially error prone because it might result in unintentional feature interferences. Thus scenarios where an unintentional feature interference might occur can be derived from a task model by finding tasks that have *no* temporal dependencies[1]. The basic assumption is that in a system all features might have potential unintentional feature interactions. In the context of AIS this is mostly true because most features use the same resources, such as communication devices.

So far, we have seen that task models explicitly model intentional feature interaction scenarios and thereby implicitly model potential unintentional feature interaction scenarios. Test cases that involve feature interaction are traces that execute a number of features. But not all valid system traces involve feature interactions. We use task models to express at an appropriate abstraction level:

- how tasks are composed of subtasks and
- how the executions of tasks constrain each other.

In the previous section 3.2 we gave an overview on reasons for feature interferences. In the following, we discuss how these issues can be addressed using test case generation based on task models.

- *Conflicting goals:* are feature interferences in which features with the same preconditions but incompatible goals are in conflict. Task models explicitly define feature interactions. During the creation of the task model, one must think about potential interactions between different features. Hence, chances are high that one identifies conflicting goals during the creation of the task model. If this is not the case, conflicting goals still can be identified during testing, when tests are generated that cover unintentional feature interaction scenarios. These scenarios might reveal faults in the system resulting from the conflicting goals.
- *Competition for resources:* Scenarios in which multiple tasks are active at the same time often lead to resource conflicts. These scenarios are explicitly described in a task model. Hence, test cases can be generated that

[1] At least no such temporal dependency is included in the model, because unintentional feature dependencies are not modeled explicitly

cover these scenarios. However, task models do not specify resource or data dependencies. Thus it is not possible to incorporate such information during test generation. Resource and data dependencies are added during test case instantiation, which is explained in the following chapters in more detail. It is reasonable to execute the same test case multiple times with different resource configurations.

- *Changing assumptions on features:* often result from new features that are added. Describing features in a task model and generating tests from the model enables the systematic and continuous test of features and their interactions. Hence, faults that result from changing boundary conditions are likely to be identified.
- *Design evolution:* When new features are introduced to a system, a task model is a means to define the dependencies to existing features. Furthermore, when a new feature is introduced, it is possible to systematically select scenarios that test the interaction or the non-existence of an interaction between the new feature and the existing ones.
- *Timing and race conditions:* Task models define no timing. Thus it is not possible to generate test cases that cover timing critical scenarios. However, scenarios in which multiple tasks interfere are often timing critical. The latter can be covered by test case generation from task models. Nevertheless, it is important to include timing during the test case instantiation, for example, by performing the same test case multiple times with different timings.

To conclude, the main advantage of task models is that they describe features and their dependencies in a concise and explicit way. Thus task models enable the systematic coverage of critical feature interaction scenarios by test case generation. This section gave an introduction into task models based on traces and showed how task models can be used to model feature interactions. In the next section we present related work in the field of task and feature modeling. Then we introduce our task modeling language *TTask* and define the semantics of task hierarchies and task dependencies in more detail.

4.2 Related Work

In this section we introduce related work from the area of feature interaction and task modeling. Furthermore, we compare task models to common modeling languages, such as state machines and activity diagrams.

4.2.1 Modeling Feature Interaction

In current research we find several approaches for the detection of feature interaction [FN03] as a part of the specification process. Calder et al. provide a comprehensive review of different approaches that cope with feature interaction in [CKMRM03]. Most approaches focus on the detection of unintentional feature interferences that result from incompatible features. The basic idea in these approaches is to model features and their dependencies. Based on such a feature model, techniques, such as model checking, are used to analyze models in order to find feature interferences. However, these approaches only find feature interferences from modeled feature interactions. From our experience, feature interferences, especially the unintentional ones, are often caused by faults in the actual implementation, such as race conditions. Therefore, we chose to use test case generation in order to detect feature interferences in the implementation as well. However, using test case generation it is not possible to detect feature interactions during specification, which can be accomplished by the previously mentioned approaches of feature interference detection. In the remainder of this section we give an overview approaches for feature interference detection in different domains.

Feature interaction has first been described for telecommunication system. For example, Kelly et al. [KCK94] use specification and description language (SDL) models to describe feature interactions. It is based on developing models of feature behavior using the language SDL[CCI84]. Their work focuses on analyzing SDL specifications in order to detect unintentional feature interferences. Their approach is to model interactions between different features and to manually simulate these models to discover feature interferences. However, they are lacking means to automatically derive feature interaction scenarios which is required for test case generation.

Today there are other domains were feature interactions occur as well. One example is feature interactions in web services. Weiss et al. employs the User Requirements Notation [Amy03] in [WE04, WEL07] to model and detect feature interactions. URN is comprised of two complementary notations: the Goal-oriented Requirements Language (GRL)[2], and the Use Case Maps (UCMs) notation[BC96]. GRL is used to model business goals, non-functional requirements, design alternatives, and design rationales. UCMs allow the description of functional requirements in the form of causal scenarios. The approach detects feature interactions by identifying conflicts in the goals that can be achieved by different features. This approach is similar to ours in using a high level goal-oriented notation to model feature interactions. However, UCMs are a semi-formal notation which does not provide means to formally analyze models or to generate test cases. Amyot et al. overcome this problem by manually transforming UCMs into LOTOS specifications. They applied UCMs [Buh98] to model feature interactions and to generate test cases [ACG$^+$00, ALW05]. They propose guidelines for the manual translation of UCMs into LOTOS specifications. The resulting LOTOS specification is used to generate test cases. They introduce three different scenario selection criteria that cover structural properties of UCMs. The manual translation into LOTOS is necessary due to the informal nature of UCMs. However, the manual transformation of UCMs into LOTOS is potentially error prone and requires a substantial knowledge of LOTOS. Furthermore, only a subset of the language constructs of UCM are translated into LOTOS. An approach where a specification must be transformed manually into a formal language is not desirable in our context. Our goal is to provide a modeling language that can be used by testers without formal background.

[2] http://www.cs.toronto.edu/km/GRL/

Zheng et al. propose in [ZZK07] their approach of test case generation for testing web service collaborations. They model service collaborations using the Business Process Execution Language (BPEL). BPEL is a semi-formal flow language that enables modeling of concurrency and hierarchy. They propose operational execution semantics for BPEL in order to use model checking for test case generation. They generate test cases using structural test selection criteria, such as state and transition coverage, as proposed in [HCL$^+$03]. Furthermore, they apply coverage criteria for data flow dependencies that were proposed by Hong et al. in [HCL$^+$03]. Their approach of test case generation is similar to the one we introduce in this thesis. They use the domain-specific language BPEL, to model their system, which corresponds to our approach of using task models. They also transform these models into model checker code in order to generate test cases. The main difference to our approach is that they use BPEL, which is a language that is focused on the specification of web services. Nevertheless, the goal of both, BPEL and task models, is to model activity flows in a system. However, we want to explicitly cover feature interaction scenarios, which are not covered by their test selection criteria.

Metzger et al. present in [Met04, MW03] an approach to detect feature interactions in embedded control systems with the example of building automation. They introduce a formal product model that is used as a basis to model feature interaction. The product model comprises a hierarchical, structural model of the system reaching from the requirements level to the implementation level. The detection of feature interactions is performed by identifying dependencies between functional needs. Interestingly, they specify functional needs in the form of tasks, which corresponds to our idea of an abstraction to model features. They stepwise refine these task into corresponding control objects using aggregation, instantiation, and realization dependencies. They describe only static dependencies therefore it is not possible to derive test cases that test feature interactions at runtime.

Rittmann describes in her thesis [Rit08] a methodology to design the usage behavior of multi-functional systems with a means to model services and their interactions. She proposes the basic service dependencies RESET, ENABLE, DISABLE, INTERRUPT, and CONTINUE as means to describe service interactions. However, she does not provide a formal definition of the semantics of the service dependencies which inhibit their formal analysis and test case generation. Nevertheless, the work provides an approach to model feature interactions and shows how this can be included in the requirements engineering process.

4.2.2 Task Modeling Notations

The concept of task analysis and modeling as a means to analyze and describe user interfaces is not new. In the following we give a short introduction into task analysis and modeling with a strong emphasis on formal task modeling approaches. A comprehensive overview on concepts, methods and languages of task modeling can be found in [DS03].

Hierarchical task analysis (HTA) was one of the first methods of task analysis, developed at the University of Hull in the late 1960s by Annett and

Duncan. Their goal was to create a rational basis for understanding the skills required in complex non-repetitive operator tasks, especially process control tasks such as those found in steel production, chemical and petroleum refining, and power generation [DS03]. HTA describes task models in terms of three main concepts: tasks, task hierarchy, and plans. Tasks are recursively decomposed into subtasks to a point where subtasks are allocated either to the user or the user interface, thus becoming observable. The task hierarchy statically represents this task decomposition. A plan specifies informally an ordering in which subtasks of a given task could be carried on.

Task models have been applied in different contexts. Stanton and Young apply task models in cognitive psychology to improve the understanding how users may interact with a given interface [SY98]. They are also used for task planning and allocation, in order to assess task workload, to plan and to allocate tasks to users in a particular organization [KA92]. In these areas task modeling has the goal to understand and structure the user interface of a system.

In software engineering, the goal of task analysis is to provide engineering models of human performance [JK96b]. In the ideal, such models produce a useful qualitative description of how the user uses a computer system to perform a task, particularly at an earlier stage in the development process than prototyping and user testing [JK96a].

The used task modeling notations are often informal descriptions without a formal foundation. In order to use task models during development of user interfaces there are several approaches that introduce task modeling notations for the early formal description of tasks and their relationships, such as MDL [Sti99], UAN [HSH90], DIANE+ [TB96], and the Concur Task Tree Notation [Pat99]. The goal of these notations is to support the design process of user interfaces by providing a precise description of tasks and their dependencies as well as the ability to generate development artifacts such as code or prototypes. In the following, we introduce the Concur Task Tree Notation in more detail.

Concur Task Trees

The concur task tree (CTT) notation has been developed by Paternò et al. [Pat99, PMM97]. Their goal is to enable the formal modeling of user interfaces in an early development phase. CTT are based on five concepts: tasks, objects, actions, operators, and roles. Figure 4.3 shows a CTT that describes the tasks of three features: Seat Control, CD and Incoming Call. The task model has been created with the CTT modeling environment CTTE [MPS02]. In the CTT notation, task models are trees where each node represents a task and where the descendants of a task represent its sub tasks. Leaf tasks are called basic tasks and represent atomic actions. CTT supports four different types of tasks depending on the involved actors:

User Tasks: are entirely performed by the user.

Application Tasks: are completely executed by the system.

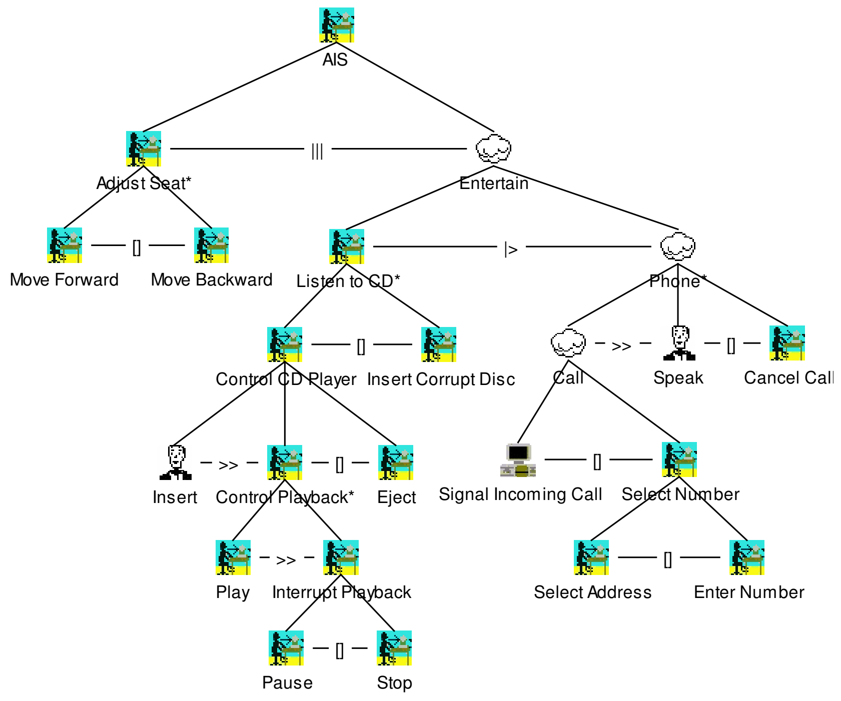


Fig. 4.3. Concur Task Tree.

Interaction Tasks: are performed by the user interacting with the system.

Abstract Tasks: are complex tasks, which are divided in different subtasks.

Temporal operators are used to link sibling tasks on the same level of decomposition. The temporal operators are defined based on LOTOS [BB87]. They connect two adjacent sibling tasks and apply for all subtasks. For example, the task Phone in the case study suspends all subtasks of the task CD. In detail there are the following kinds of temporal relations:

Choice ([]): It is possible to choose between a set of tasks. If one task is active the other tasks from the set can not be selected. For example the user can stop or pause the playback.

Order Independence (| = |): Both tasks have to be performed, but if one task has started, it has to be finished, before the other can start.

Concurrent (|||): Both tasks can be executed concurrently.

Concurrent with info exchange (|[]|): Both tasks can be executed concurrently, but have to be synchronized in order to exchange information.

Disabling ([>): When one task is finished, it disables another task. For example if a disc is ejected, the task play disc is disabled.

Suspend/Resume ($|>>$): The execution of a task $t1$ interrupts the execution of another task $t2$, which resumes after $t1$ has finished.

Enabling ($>>$): If a task finishes, it enables the execution of another task. For example the task insert disc enables the task play.

Enabling with info exchange ($[]>>$): If a task finishes, it enables the execution of another task with additional information exchange. For example, if a CD is inserted, the playback of the CD is enabled and the track list is exchanged.

Iteration ($*$): The task can be performed multiple times.

Actions and objects are specified for each basic task. Objects could be perceivable objects or application objects. Application objects are mapped onto perceivable objects in order to be presented to the user. An interesting feature of CTT is that both input actions and output actions associated with an object are specified. The last modification made to CTT was to add the concept of platform in order to support multi-platform user interface development.

Our first approach of test case generation from task models used the CTT notation [Ben07]. When we tried to describe several features comprising of multiple intentional interactions, CTT did not provide the means of encapsulating feature-specific tasks due to the constraint that temporal operators are only allowed between sibling tasks. Furthermore, CTTs have a dedicated operator to describe the concurrent execution of two tasks. An AIS, for example, has a large number of tasks that can be performed concurrently which therefore must all be explicitly defined. Therefore, we decided to create a new task modeling notation that provides a formal foundation which can be used to generate test cases and that provides a concise way to model large systems such as AIS.

4.2.3 Test Case Generation Approaches

In this section we compare common modeling languages that are used for test case generation with task models. First we compare task models and finite state machines. Finite state machines are a widely used technique for the specification of reactive systems. They describe the possible states of a system and the transitions between these states. They are a common technique for test case generation. The main difference between task models and finite state machines is that task models describe possible sequences of activities in contrast to finite state machines that describe possible sequences of state transitions. However, a task model can be described by an automaton by encoding the different modes of a task in states. Figure 4.4 shows a task model and a state machine. Both describe a *suspend/resume* dependency between two tasks. The state machine model is far more complex than the task model. The complexity of the state machine increases even more when additional tasks with *suspend/resume* dependencies are added, which would soon result in the unreadability of the state machine. Hence, modeling larger systems that comprise a large number of features is not feasible with state machines.

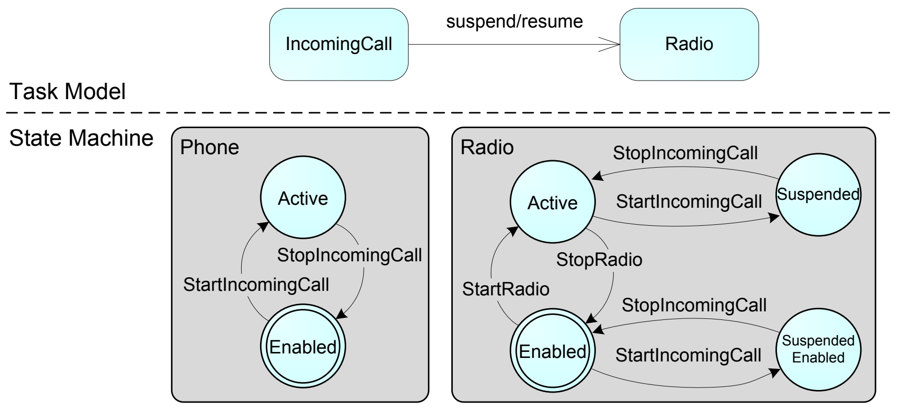


Fig. 4.4. Task model versus state machine.

Chi and Hao introduce in [CH07] an approach for generating test cases that cover feature interactions from finite state machines. They generate test sequences to detect feature interactions in a complex feature-rich communication system. Their approach shows that it is possible to generate test cases that cover feature interactions. However, from our experience state machines are not the right abstraction to model features interactions, because they soon result in large and unmaintainable models.

A more appropriate abstraction to model feature interactions are activity diagrams that are part of the unified modeling language (UML). They focus on activities instead of state transitions by describing the sequential or concurrent control flows of activities. They are typically used for modeling the logic captured by a single use case or usage scenario. In addition to task models they support elements such as conditions and decisions. Nóbrega et al. describe in [NNC05] the mapping of CTT to UML 2.0 Activity diagrams. The mapping resulted in incomprehensibly large activity diagrams because the native temporal operators in a CTT were not supported and therefore had to be reproduced using complex constructs. However, these large activity diagrams show that task models are able to express task dependencies in a more compact and expressive manner.

Activity diagrams are used for test case generation as well. Mingsong et al. present in [MXX06] an approach that supports the test case generation from UML activity diagrams. They support structural test selection criteria, such as activity coverage and transition coverage. The test case generation is performed by generating random test cases until the generated test suite fulfills certain structural coverage criteria. This approach is only applicable when the effort of test case execution is low, which is not true for AIS. Hence, it is necessary to be able to generate a minimal set of test cases that cover critical scenarios.

To summarize, task models can be expressed by state machines and activity diagrams. However, the advantage of task models is their compact notation that enables the specification of larger systems in comparison to general-purpose modeling languages such as state machines and activity diagrams. Another important advantage is that task models support different operators to describe temporal dependencies between tasks. This enables the definition of specific test selection criteria that cover interaction scenarios based on these dependency types.

4.3 The *TTask* Language

We have seen in the first part of this chapter that task models are an appropriate technique to describe feature interactions. What is needed is a task modeling notation that combines a formal foundation with a simple graphical syntax in order to enable test case generation for testers with no background in formal methods. In the remainder of this chapter, we present such a modeling language: *TTask*.

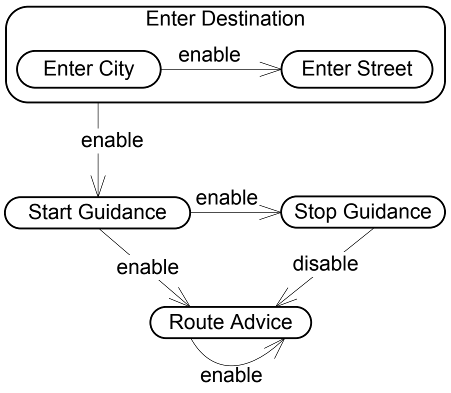


Fig. 4.5. Route guidance task model.

A *TTask* model consists of *task hierarchies* and *temporal dependencies*. *Task hierarchies* enable the specification of complex *tasks* by decomposing them into their *subtasks*. *TTask* supports three different temporal dependencies: *enable*, *disable* and *suspend*. These dependencies can be used to describe, for example, when the execution of a task enables or disables another task. Based on these temporal dependencies, a *TTask* model implicitly describes the space of all possible sequential task executions. Figure 4.5 shows a sample task model that describes the necessary tasks to enter a navigation destination and to control the route guidance for an AIS. A task is illustrated as a rounded rectangle in a *TTask* diagram. Task hierarchies are depicted by drawing *subtasks* inside of their parent tasks. In the example in Figure 4.5 the task Enter Destination is a composite task that is divided into two subtasks Enter City and Enter Street. The *temporal dependencies* between two tasks are symbolized by directed edges. For example, the tasks Enter City and Enter Street have an *enable* dependency, which means that the execution of Enter City enables the execution of Enter Street. Note that *TTask* does not differentiate between tasks that are started by the environment (e.g. Enter City) and tasks that are started by the system (e.g. Route Advice).

Formally a *TTask* model M is a quadruple $M = (T, S_0, \Delta, \Phi)$, where:

- T is a set of tasks $T = \{t_1, ..., t_n\}$, where each $t_i \in T$ is a unique task identifier.
- The task hierarchy is described by the subtask function Φ that maps a task to its subtasks $\Phi(T) \rightarrow \{T\}$. For Φ must hold that:
  - there are no cycles in the task hierarchy.
  - $\forall t \in T.\ \exists t_1 \in T.\ \exists t_2 \in T.\ t \in \Phi(t_1) \wedge t \in \Phi(t_2) \rightarrow t_1 = t_2$
- Δ is a set of temporal task dependencies where each dependency $\delta \in \Delta$ is a triple $\delta = (t_i, t_j, d)$, where $t_i, t_j \in T$ and $d \in \{enable, disable, suspend\}$.
- $S_0 = \langle m_1, ..., m_n \rangle$ is the initial *task model state* that describes for each task $t_i \in T$ its initial mode $m_i \in \{enabled, disabled\}$.

We refer to a task t that has no subtasks $\Phi(t) = \emptyset$ as *atomic task* and otherwise as *composite task* $\Phi(t) \neq \emptyset$. In the task model in Figure 4.5 the tasks Enter City and Enter Street are *atomic tasks* (among others). Atomic tasks represent actual interactions between system and environment whereas composite tasks represent the composed interactions of their subtasks. In the example, Enter Destination is performed by the sequential execution of its two subtasks Enter City and Enter Street.

Task dependencies describe the temporal relations between tasks based on the task hierarchy: a task dependency holds for a task and its subtasks. For example, in the task model in Figure 4.5, the *enable* dependency between Enter Destination and Start Guidance holds for the subtasks of Enter Destination as well. First we introduce the definition of a task's *closure* in order to define the dependencies between two tasks. The closure of a task t is the smallest set that contains the task and all its ancestors:

$$closure(t) = \begin{cases} t \cup closure(t_p) & , \exists t_p \in T.\ t \in \Phi(t_p) \\ t & , otherwise \end{cases} \tag{4.1}$$

Two task t_1 and t_2 have a dependency $d \in \{enable, disable, suspend\}$ if they or one of their ancestors have a corresponding dependency:

$$\exists t_{p1} \in closure(t_1).\ \exists t_{p2} \in closure(t_2).\ \exists \delta \in \Delta . \delta = (t_{p1}, t_{p2}, d)$$

The goal of a task model is to describe the temporal relations between different tasks by defining how the execution of a task influences other tasks. For example, the execution of a task enables or disables the execution of another task. A *task mode* defines whether a task can be executed or not. A task can only be executed if its mode is *enabled*. Task modes describe the current state of a task, based on its *initial mode* and on the history of already executed tasks. A task mode change results from the execution of tasks that are related by a temporal dependency. The set of possible task modes is defined by:

$$\text{MODES} = \{enabled, disabled, active, suspended, \\ suspended\_enabled, suspended\_disabled\} \tag{4.2}$$

The task modes are in detail:

- **enabled:** The task is ready for execution. It is either initially enabled or it has been enabled by another task.
- **active:** The task is currently being executed.
- **disabled:** The task is disabled and cannot be executed. It is either initially disabled or it has been disabled by another task.
- **suspended:** The task has been active and is being suspended by another task. A task t is suspended as long as any other task t_S that has a *suspend* dependency $\delta = \langle t_s, t, suspend \rangle$ to t is active.
- **suspended_enabled:** The task is both enabled and suspended. Hence, the task cannot be executed until it is resumed by the suspending task.
- **suspended_disabled:** The task is both disabled and suspended. In order to enable the task, it must be enabled and resumed by other tasks.

In *TTask*, the actual interaction between environment and system is not described, instead these interactions are represented by the execution of tasks. A task is executed by first starting and then stopping it. When a task is started, it changes its mode from *enabled* to *active* and when it is stopped it changes its mode from *active* to *disabled*. The start of a task represents the beginning of a specific interaction between system and environment and the stop of a task represents the end of this interaction between system and environment. The environment and system are not explicitly modeled in a *TTask* model, instead they are combined and represented by the Executor that is responsible for starting and stopping tasks.

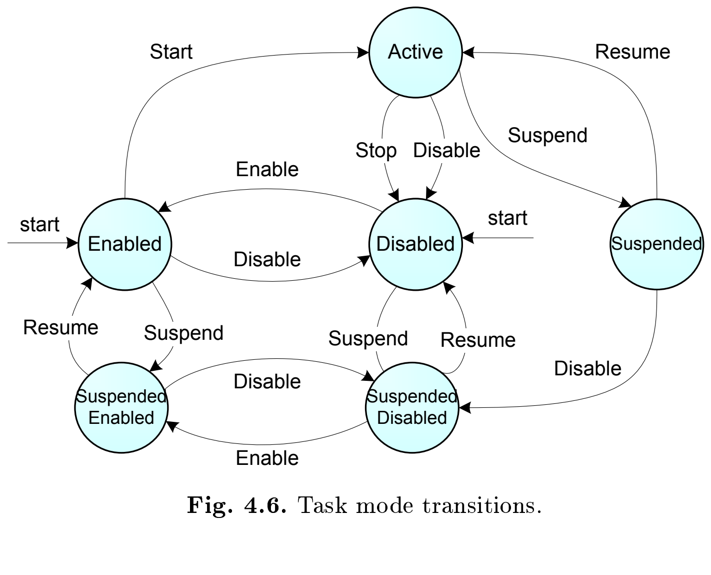


Fig. 4.6. Task mode transitions.

When a task is performed, the global task model state changes. Figure 4.6 shows a state diagram that describes these task mode transitions in the form of a mealy machine. The state machine shows the resulting task mode transitions when the executor starts or stops a task. These are represented by the inputs start and stop. Furthermore, the state machine shows the task mode changes

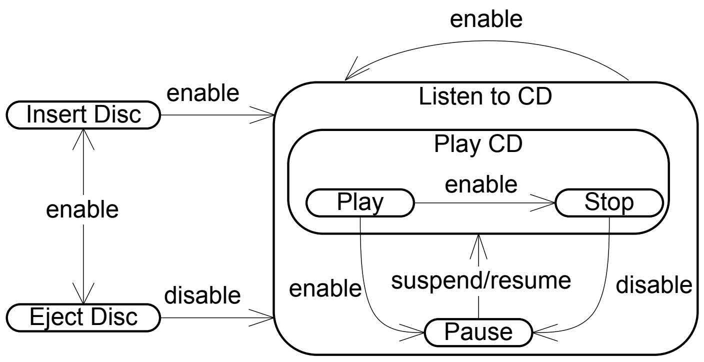


Fig. 4.7. Task model for the CD feature.

that result from the task dependencies: the state transitions for the inputs enable, disable, suspend, and resume. Initially either the state enabled or the state disabled is active. There are two transitions from the task mode *active* to the mode *disabled*. They differ based on the initiator of the task mode switch. The first transition for the input *stop* is fired when the task is completed and therefore stops. The other transition for the input *disable* is fired when the task is disabled by another task that becomes active.

In the following we introduce the temporal task dependencies in more detail based on the example in Figure 4.7. The task model describes the tasks of a simple CD player application. The depicted tasks are:

1. **Insert Disc:** The user performs this task by inserting a disc into the player.
2. **Eject Disc:** The user performs this task by pressing the eject button.
3. **Play:** The user performs this task by pressing the play button.
4. **Stop:** The user performs this task by pressing the stop button.
5. **Play CD:** This task represents the activity of the CD player, when it plays a disc. It is started when the task Play is started and is stopped when the task Stop is finished.
6. **Pause:** The user performs this task by pressing the pause button twice: first to pause the CD player and then to resume the playback. While this task is performed, the task Play CD is suspended.
7. **Listen to CD:** This task describes the overall behavior of the CD player. It can be performed iteratively.

The task model in Figure 4.7 uses all three task dependencies *enable*, *disable*, and *suspend* in order to describe the behavior of a CD player:

enable: The *enable* dependency expresses that one task is enabled when another task is finished. This allows the definition of task sequences, where one task is a precondition for another task. In Figure 4.7 the task Insert Disc enables the task Eject Disc and vice versa. When Insert Disc finishes, the task Eject Disc is enabled and can be performed. The same applies to Listen to CD which is also enabled by Insert Disc. Temporal task dependencies hold for a task and its subtasks. Hence, all *initially enabled* subtasks of Listen to CD are enabled as well. A subtask is *initially enabled* if it has **no** incoming *enable* dependency from any sibling task or any of its sibling tasks' descendants. The task Listen to CD can be performed iteratively. This is modeled by an *enable* dependency to itself. Hence, when the task Listen to CD is stopped, it changes its mode to *disabled* and subsequently enables itself, which results in its new mode being *enabled*.

disable: The situation that the execution of a task disables another task can be described with the *disable* dependency. In Figure 4.7 the task Eject Disc disables the task Listen to CD. When the task Eject Disc is started, the task Listen to CD and all of its subtasks become disabled, even if the task Listen to CD is *active*. The latter is depicted in Figure 4.6 by the transition for the input disable between the states active and disabled. As a consequence Listen to CD and all its child tasks cannot be performed until they are re-enabled by Insert Disc.

suspend: Interruption scenarios can be described with the *suspend* dependency. A task is suspended during the execution of a suspending task and it is resumed when the suspending task is finished. The task Pause in Figure 4.7 suspends the task Play CD when it starts and resumes it after Pause is stopped. If a task is suspended by multiple active tasks it is not resumed until all suspending tasks are finished.

In the following, we introduce the notion of a *task sequence* as a means to describe sequential task executions in a task model. Figure 4.8 shows three different task model states of the same task model. Different task modes are represented by different colors. The first task model state shows the task modes before the task Enter City is started and the second task model state shows the task modes after the task Enter City is started. The last task model state shows the resulting task modes when Enter City is completed and therefore is stops. When an atomic task is currently performed, thus its mode is *active*, all parent tasks are *active* as well, which is the case for the composite task Enter Destination in the second task model state. Parent tasks remain active as longs as one of their subtasks is *enabled* or *active*. In the example, Enter Destination gets disabled after the task Enter Street has been performed.

Formally, a task model state S of a given task model $M = (T, S_0, \Delta, \Phi)$, is defined as an ordered set of task modes:

$$S = \langle m_0, ..., m_n \rangle$$

We define a mapping function *mode* that maps a task to its mode in a task model state:

$$mode : S \times T \rightarrow MODES$$

We use the following abbreviation for *mode* to select the mode m_i of a task t_i from a task model state S:

$$S[t_i] = m_i$$

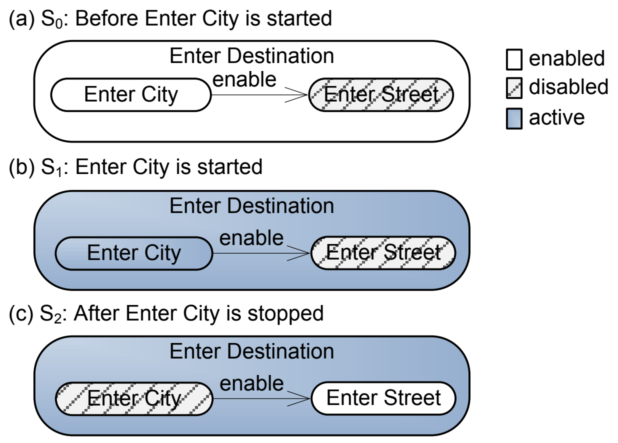


Fig. 4.8. Task modes before and after Enter City is executed.

Thus Figure 4.8 shows three task model states S_0, S_1, and S_2 that results from the execution of the task Enter City. Additionally, the table in Figure 4.9 shows all five task model states that result from the sequential execution of the tasks Enter City and Enter Street. In this table, we see that the composite task Enter Destination changes its mode to *disabled* after Enter Street is performed. In general we refer to such a sequence of task model states as a *task sequence.*

	Enter Destination	**Enter City**	**Enter Street**
S0	enabled	enabled	disabled
S1	active	active	disabled
S2	active	disabled	enabled
S3	active	disabled	active
S4	disabled	disabled	disabled

Fig. 4.9. Task sequence.

In the remainder of this section, we introduce the execution semantics of a $TTask$ model. In order to simplify the definition of the execution semantics, we define the update function $S[t_i, m] = S'$ that changes the mode of a task t_i in a task model state S to the new mode $m \in MODE$:

$$S[t_i, m][t_j] = \begin{cases} m & , t_i = t_j \\ S[t_j] & , otherwise \end{cases}$$

A *task sequence* TS is a sequence of task model states

$$TS = \langle S_0, ..., S_p \rangle, \text{ where } S_i = \langle m_0, ..., m_n \rangle$$

for which holds:

- S_0 is the initial task model state.
- Three subsequent task model states S_i, S_{i+1}, and S_{i+2} result from the sequential start and stop of an enabled atomic task t_j

$$\Phi(t_j) = \emptyset \wedge S_i[t_j] = \mathit{enabled}$$

beginning at the initial task model state S_0:

$$\mathit{start}(S_i, t_j) = S_{i+1} \wedge \mathit{stop}(S_{i+1}, t_j) = S_{i+2}$$

where:

$$\mathit{start} : \langle \mathit{MODE} \times \mathit{MODE} \times ...\rangle \times T \rightarrow \langle \mathit{MODE} \times \mathit{MODE} \times ...\rangle$$

describes the resulting task model state when a task is started and

$$\mathit{stop} : \langle \mathit{MODE} \times \mathit{MODE} \times ...\rangle \times T \rightarrow \langle \mathit{MODE} \times \mathit{MODE} \times ...\rangle$$

describes the resulting task model state when a task stopped.

When a task is started, its mode is changed to *active*. If the task has a parent task, the parent task is started as well. Subsequently, all tasks that are related by *suspend* and *disable* dependencies to the started task are suspended and disabled respectively. The start of a task is defined by:

$$\mathit{start}(S, t_i) = \mathit{suspendTasks}(\mathit{disableTasks}(\mathit{startParent}(S[t_i, \mathit{active}], ..., t_i))) \tag{4.3}$$

The start of the parent task is the smallest set for which holds:

$$\mathit{startParent}(S, t_i) = \begin{cases} \mathit{start}(S, t_p) & , \exists t_p \in T.\ t_i \in \Phi(t_p) \\ S & , \mathit{otherwise} \end{cases} \tag{4.4}$$

Using the definition of a closure we can define the set of disabled tasks by a given task t by:

$$\mathit{disabledBy}(t) = \{t_d | \exists t_p \in \mathit{closure}(t_d).\ \exists \delta \in \Delta.\ \delta = (t, t_p, \mathit{disable})\} \tag{4.5}$$

The mode of the disabled tasks is changed depending on their current mode. For example, an *enabled* task's mode is changed to *disabled*. The resulting task modes for disabled tasks are:

$$\mathit{disableTasks}(S, t_i)[t_d] = \begin{cases} \mathit{suspended\_disabled} & , t_d \in \mathit{disabledBy}(t_i) \wedge (S[t_d] = \mathit{suspended\_enabled} \\ & \quad \vee S[t_d] = \mathit{suspended}) \\ \mathit{disabled} & , t_d \in \mathit{disabledBy}(t_i) \wedge (S[t_d] = \mathit{enabled} \\ & \quad \vee S[t_d] = \mathit{active}) \\ S[t_d] & , \mathit{otherwise} \end{cases} \tag{4.6}$$

All tasks that are related by the *suspend* dependency are suspended. For example, a task's mode is changed from *enabled* to *suspended_enabled*. The set of suspended tasks by a given task t is:

$$\mathit{suspendedBy}(t) = \{t_d | \exists t_p \in \mathit{closure}(t_d).\ \exists \delta \in \Delta.\ \delta = (t, t_p, \mathit{suspend})\} \tag{4.7}$$

The resulting task modes for suspended tasks are:

$$suspendTasks(S,t_i)[t_d] = \begin{cases} suspended & , t_d \in suspendedBy(t_i) \ \wedge \ S[t_d] = active \\ suspended\_enabled & , t_d \in suspendedBy(t_i) \ \wedge \ S[t_d] = enabled \\ suspended\_disabled & , t_d \in suspendedBy(t_i) \ \wedge \ S[t_d] = disabled \\ S[t_d] & , otherwise \end{cases} \tag{4.8}$$

A task is stopped by changing its mode to *disabled*. Subsequently, all tasks that are related by the *enable* dependency are enabled. If no subtask of the parent task is enabled or active, the parent task is stopped as well. Finally all tasks that are related by *suspend* are resumed. When a task t_i is stopped the resulting task model state S' is defined by:

$$stop(S,t_i) = resumeTasks(stopParent(enableTasks(S[t_i, disabled], ..., t_i))) \tag{4.9}$$

When a task stops, it enables all tasks that are related by the *enable* dependency and their *initially enabled* subtasks if they are not disabled by another active task. *Initially enabled* subtasks have no incoming *enable* dependency from a sibling task or a sibling task's descendant. The siblings of a task and their descendants are defined by:

$$siblings(t) = \{t_i | \exists t_p \in T.\ t \in \Phi(t_p) \ \wedge \ t_P \in closure(t_i) \ \wedge \ t \notin closure(t_i)\} \tag{4.10}$$

The *initially enabled* subtasks of a task t are defined by:

$$\begin{aligned} initiallyEnabled(t) = \{t_i \in T | t \in closure(t_i) \\ \wedge \ \exists t_p \in closure(t_i).\ \exists \delta \in \Delta.\ \nexists t_e \in siblings(t_i). \\ t \in closure(t_p) \ \wedge \ \delta = (t_e, t_p, enable)\} \end{aligned} \tag{4.11}$$

The set of enabled tasks by a given task t is:

$$\begin{aligned} enabledBy(t) = \{t_d \in T | \exists t_p \in closure(t_d).\ \exists \delta \in \Delta. \nexists t_y \in T. \\ \delta = (t, t_p, enable) \wedge \ t_d \in initiallyEnabled(t_p) \\ \wedge \ S[t_y] = active \wedge t_p \in disabledBy(t_y)\} \end{aligned} \tag{4.12}$$

The modes of the enabled tasks change as follows:

$$enableTasks(S,t_i)[t_d] = \begin{cases} enabled & , t_d \in enabledBy(t_i) \ \wedge \ S[t_d] = disabled \\ suspended\_enabled & , t_d \in enabledBy(t_i) \ \wedge \ S[t_d] = suspended\_disabled \\ S[t_d] & , otherwise \end{cases} \tag{4.13}$$

The task model state after stopping the parent task is defined by:

$$stopParent(S,t_i) = \begin{cases} stop(S,t_p) & , \exists t_p \in T.\ t_i \in \Phi(t_p) \\ & \quad \wedge \forall t_c \in \Phi(t_p).\ S[t_c] = disabled \vee S[t_c] = suspended\_disabled \\ S & , otherwise \end{cases} \tag{4.14}$$

The set of resumed tasks by a given task t where no other suspending task is active is:

$$\begin{aligned} resumedBy(t) = \{&t_d | t_d \in suspendedBy(t) \\ &\wedge \nexists t_x \in T.\ t_d \in suspendedBy(t_x) \wedge S[t_x] = active\} \end{aligned} \tag{4.15}$$

Finally, the resulting task modes for suspended tasks are:

$$resumeTasks(S, t_i)[t_d] = \begin{cases} active & , t_d \in resumedBy(t_i) \ \wedge \ S[t_d] = suspended \\ enabled & , t_d \in resumedBy(t_i) \ \wedge \ S[t_d] = suspended\_enabled \\ disabled & , t_d \in resumedBy(t_i) \ \wedge \ S[t_d] = suspended\_disabled \\ S[t_d] & , otherwise \end{cases} \tag{4.16}$$

The last paragraphs described the semantics of a *TTask* model. The most important points are:

- When a task is started, it changes its mode from *enabled* to *active*. When it is stopped, it changes its mode from *active* to *disabled*.
- A composite task is started when it is *enabled* and one of its subtasks is started. It is stopped, when no subtasks are either *active* or *enabled*.
- An *active* atomic tasks must be stopped before another atomic task can be started. Hence, there are never multiple atomic tasks *active* at the same time. However, composite tasks can be *active* at the same time.
- When a composite task is enabled, all subtasks that have no incoming *enable* dependency from a task that is part of the same task hierarchy, are initially enabled.

To sum up, *TTask* enables the definition of hierarchical tasks and their temporal dependencies. The goal is to use *TTask* to describe feature interactions. To achieve this we have to provide a definition of features. We stated earlier that a task describes a feature. Hence, the definition of a feature is straightforward. For a given task model $M = (T, S_0, \Delta, \Phi)$, a feature is defined by a set of tasks:

$$f \subset T$$

Based on the definition of a feature we can formally define feature interactions as dependencies between tasks that belong to different features. Let $D(TASK \times TASK) \rightarrow \{\Delta\}$ be the set of all temporal dependencies between two tasks t_1 and t_2:

$$D(t_1, t_2) = \{d \in \Delta : \exists t_{p1} \in closure(t_1).\ \exists t_{p2} \in closure(t_2).\ \delta = (t_{p1}, t_{p2}, d)\}$$

Two features $f_1 \subset T$ and $f_2 \subset T$, where $f_1 \cap f1 = \emptyset$, have an intentional feature interaction, iff:

$$\exists t_1 \in f_1.\, \exists t_2 \in f_2.\, D(t_1, t_2) \neq \emptyset$$

These two features have no intentional feature interaction, iff:

$$\nexists t_1 \in f_1 .\, \nexists t_2 \in f_2 .\, D(t_1, t_2) \neq \emptyset$$

In this case, when two features have no intentional interactions, there might be unintentional feature interferences during the execution of any of these feature's tasks. However, even if two features have an intentional interaction, there still might be unintentional feature interferences between them. Namely, when there are two tasks $t_1 \in f_1$ and $t_2 \in f_2$ that have no temporal dependency and if there are two tasks $t_3 \in f_1$ and $t_4 \in f_2$ that have a temporal dependency:

$$\exists\, t_1 \in f_1 .\, \exists t_2 \in f_2 .\, D(t_1, t_2) = \emptyset \wedge$$
$$\exists\, t_3 \in f_1 .\, \exists t_4 \in f_2 .\, D(t_1, t_2) \neq \emptyset$$

In that case, the tasks t_1 and t_2 might have an unintentional feature interaction, as well. This shows that it is possible to derive from a *TTask* model all scenarios that involve an intentional feature interaction and all scenarios that might involve an unintentional feature interaction. Hence, a *TTask* model can be used to generate test cases that systematically test the existence of intentional feature interactions and test the non-existence of unintentional feature interferences.

To summarize, *TTask* is a task modeling language that enables the hierarchical description of tasks and their temporal dependencies. Based on these temporal dependencies it is possible to describe the enabling, disabling and suspension of one task by another task. In contrast to other task modeling languages such as CTTs [PMM97], concurrency must not be defined explicitly, instead all tasks can be performed concurrently, as long as they are not excluded by temporal dependencies.

We do not claim that *TTask* is complete in terms of being able to model the behavior of any system. For instance, it is currently not possible to model the situation when the same task can be performed concurrently with a prior unknown number of concurrent executions. For example, when the user would be able to hold an unlimited number of phone calls at the same time.

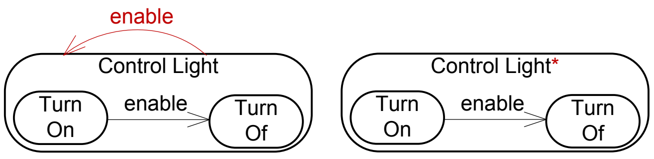
In *TTask* there are only three different task dependencies defined. Hence, the possibilities to describe the dependencies between two single tasks is limited. However, the combination of these task dependencies with hierarchical task structures enables the specification of complex behavior patterns. We were able to model large parts of the user interface of an actual infotainment system using *TTask*. The next section introduces modeling patterns for *TTask* which describe typical scenarios that have occurred in practice.

4.4 *TTask* Modeling Patterns

There are certain behavioral patterns that occur frequently in practice. An example for such a typical behavior pattern is the mutual exclusion of two tasks. Task modeling patterns are templates to describe typical behavior scenarios similar to design patterns for programming languages [GHJV95]. The two goals of this section is to firstly provide a more profound understanding of *TTask* and its capabilities by giving examples how common interactions can be described with *TTask* and secondly to introduce the extended graphical syntax of *TTask* in order to simplify the application of these modeling patterns,

4.4.1 Iterative Task Pattern

We refer to a task that can be performed multiple times as an *iterative task*. This is modeled by an *enable* dependency to itself. Thus, when the task is finished, it changes its mode from *active* to *disabled*. It then enables all tasks that are related by the *enable* dependency, including itself, resulting in its new mode being *enabled*. Thus it can be executed again. Figure 4.10 shows an example of the *iterative task pattern* where Figure 4.10(a) shows the basic realization in *TTask* and Figure 4.10(b) shows the simplified graphical syntax. The asterisk behind a task's name indicates that it is an *iterative task*.



(a) Basic realization. (b) Simplified graphical representation.

Fig. 4.10. Iterative pattern.

4.4.2 Sequential Execution Pattern

Tasks are often executed sequentially. The implementation is straightforward by using the *enable* dependency. Figure 4.11 shows an example of the *sequential execution pattern*. The composite task Select Song has three subtasks that are performed sequentially.

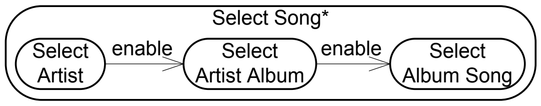


Fig. 4.11. Sequential execution pattern.

4.4.3 Order Independence Pattern

When the execution order of two or more tasks is undefined, we can use the *order independence pattern* to describe this situation. Figure 4.12 shows an example for the task Add Contact. When the user creates a new contact, the user may enter the contact's forename and surname in any order. This is modeled by a composite task with no temporal dependencies between its subtasks. The composite task is started when one of its subtasks is started. This is the case, when either Enter Surname or Enter Forename is started. It stops when all subtasks are *disabled*.

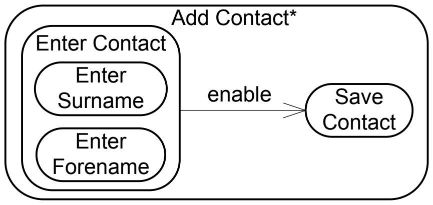


Fig. 4.12. Order independence pattern.

4.4.4 Task Interleaving Pattern

When a system can perform multiple functions concurrently, the corresponding tasks are executed interleaved. Two or more composite tasks can be performed interleaved, when they have no temporal dependencies. This is a typical situation in which an unintentional feature interaction might occur. This situation is described by composite tasks with no temporal dependencies as shown in Figure 4.13. The two tasks Control Light and Add Contact are independent of each other and therefore the atomic subtasks can be performed interleaved.

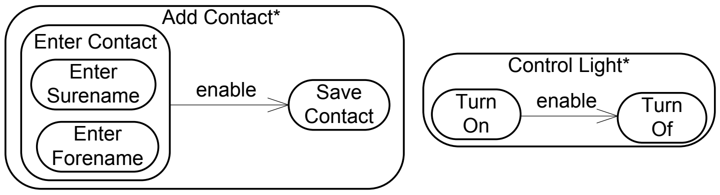
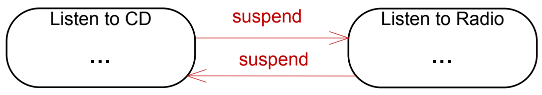


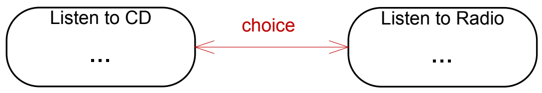
Fig. 4.13. Task interleaving pattern.

4.4.5 Mutual Exclusion Pattern

When the execution of a task excludes the execution of another task the *Mutual Exclusion Pattern* applies. For example, the tasks Listen to Radio and Listen to CD cannot be performed at the same time. This is modeled by two opposing *suspend* dependencies as depicted in Figure 4.14. Figure 4.14(a) shows the implementation and Figure 4.14(b) shows the simplified graphical syntax with only one bidirectional dependency.



(a) Basic realization.

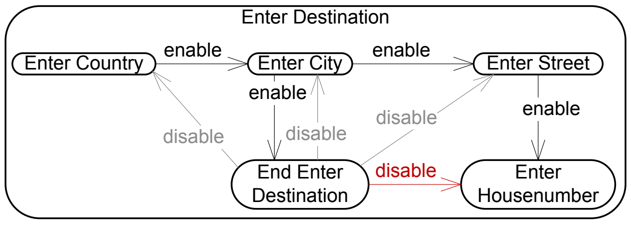


(b) Simplified graphical representation.

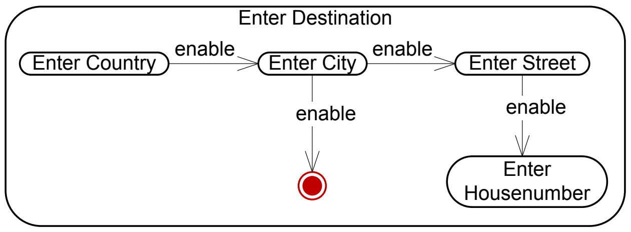
Fig. 4.14. Mutual exclusion pattern.

4.4.6 Optional Task Pattern

When a user enters a new destination into the navigation system, the user can start the route guidance after entering a city or additionally entering a street and a house number. Hence, the tasks Enter Street and Enter House Number are optional and need not to be performed in order to finish the task Enter Destination. The *Optional Task Pattern* implements this situation by adding an explicit end task. Figure 4.15(a) shows the implementation with an additional end task End Enter Destination. End Enter Destination is enabled by the task Enter City and it has a *disable* dependency to the tasks Enter Street and Enter House Number. Thus, when End Enter Destination is performed the optional tasks become disabled and therefore Enter Destination is finished. However, the introduction of an explicit end task does not correspond to the real world because the user must not explicitly end the destination input. Rather the user implicitly ends the task by starting another task. Figure 4.15(b) shows the simplified graphical syntax with an *end task*. An *end task* is symbolized by a filled circle that represents a task that has a *disable* dependency to all sibling tasks.



(a) Basic realization.

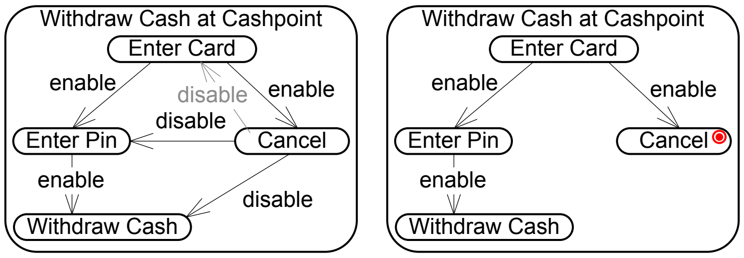


(b) Simplified graphical representation.

Fig. 4.15. Optional tasks pattern.

4.4.7 End Task Pattern

The end task pattern is related to the previous pattern. Here we have one task that explicitly ends the parent task by disabling all sibling tasks. For example, a task model for a simple cash machine. After the user enters the card, the user can always cancel the task Withdraw Cash at Cashpoint by performing the task Cancel. The Cancel task disables all other sibling tasks and thereby ends the parent task Withdraw Cash at Cashpoint. In the first task model in Figure 4.16(a) this is modeled explicitly by *disable* dependencies and in the task model in Figure 4.16(b) this is implicitly modeled by modeling Cancel as an end task which represents a *disable* dependency to each sibling task. The difference to the previous pattern is that here the end task corresponds to a task in the real world whereas in the previous pattern the end task has no corresponding task in the real world.



(a) Basic realization. (b) Simplified graphical representation.

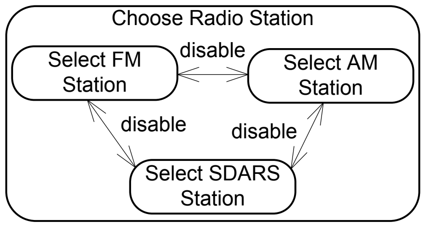
Fig. 4.16. End task pattern.

4.4.8 Selection Pattern

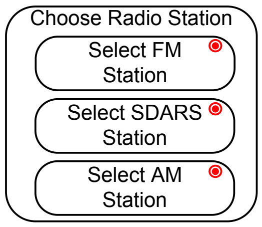
The *Selection Pattern* represents the situation where a composite task is performed when one of its subtasks is performed. For example, the user can select a radio station either by selecting an amplitude modulation (AM) station, frequency modulation (FM) station, or a satellite radio station (SDARS). Hence, the user has the choice between three different tasks in order to perform the task Choose Radio Station. In the first task model in Figure 4.17(a) this is modeled explicitly by *disable* dependencies between the subtasks and in the task model in Figure 4.17(b) this is implicitly modeled using the end task pattern.

4.4.9 Long Running Task Pattern

The execution of an atomic task can not be interleaved with the execution of another task. There are two different ways to describe long running tasks which can be executed interleaved with other tasks. The first solution is to explicitly define a start and stop task as shown in Figure 4.18(a) for the task Listen to CD. The second solution is depicted in Figure 4.18(b). The end of the long running task Listen to Radio is implicitly described by an end task.

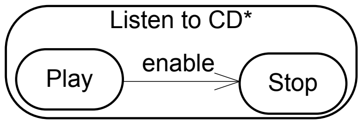


(a) Basic realization.

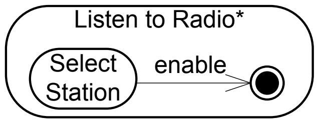


(b) Simplified graphical representation.

Fig. 4.17. Choice pattern.



(a) Long running task with start and end task.



(b) Long running task with implicit end task.

Fig. 4.18. Long running task.

4.5 Modeling Feature Interaction with *TTask*

The previous sections introduced the *TTask* notation. Furthermore, we introduced modeling patters to describe typical behavioral scenarios of a system. This section demonstrates how *direct* and *indirect feature interactions* can be described with *TTask*. Figure 4.19 shows the task model from Figure 4.7 with two additional tasks: Handle Incoming Call and Adjust Seat.

In Section 4.1 we introduced tasks as a way to represent feature-specific interaction between a system and its environment. As an example: the tasks Signal Call, Accept Call, and Reject Call describe the feature Incoming Call. With *TTask* we can describe the feature-specific behavior with these tasks and their temporal relations. Furthermore, we can describe feature interactions by temporal dependencies between tasks that belong to different features. Figure 4.19 shows an example task model that describes four features: CD Playback, Radio, Handle Incoming Call and Adjust Seat. These features are highlighted in gray.

This task model describes multiple intentional feature interactions. Direct feature interactions are task dependencies between tasks that belong to different features. For example the *suspend* dependency between the mutual exclusive tasks Listen to Radio and Listen to CD. Both tasks belong to the composite task Listen to Media which is suspended by Handle Incoming Call. Hence, the *suspend* dependency between Handle Incoming Call and Listen to Media describes two different feature interactions. These task dependencies describe the following feature interaction scenarios:

Listen to Radio $\xrightarrow{suspend}$ Listen to CD: When the user listens to the radio, the listening to a CD is suspended.

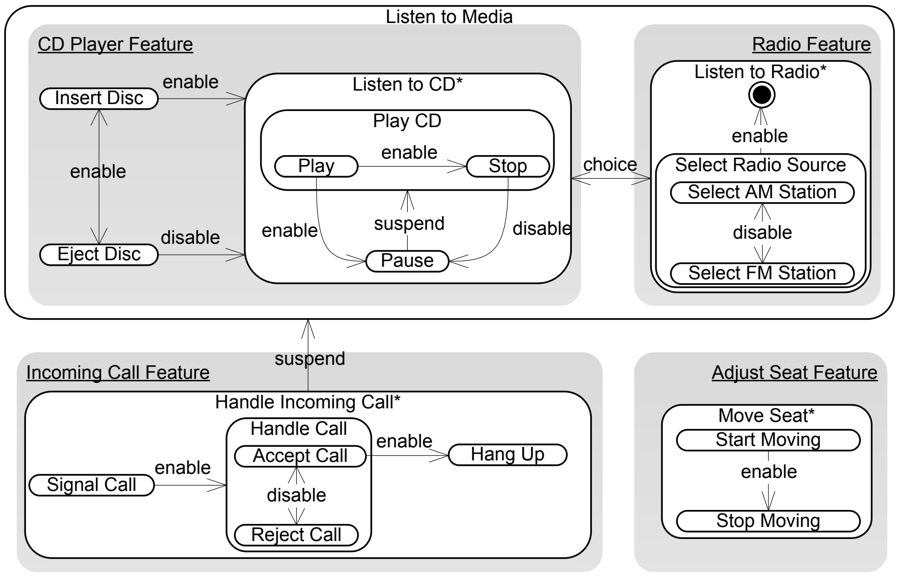


Fig. 4.19. Four features and their interactions described with *TTask*.

Listen to CD $\xrightarrow{suspend}$ Listen to Radio: When the user listens to a CD, listening to the radio is suspended.

Handle Incoming Call $\xrightarrow{suspend}$ Listen to Radio: An incoming call interrupts listening to the radio. There are two possible situations, when an incoming call can interrupt the *active* task Listen to Radio depending on the last radio task that has been performed:

1. Select AM Station $\rightarrow$ Handle Incoming Call
2. Select FM Station $\rightarrow$ Handle Incoming Call

$IncomingCall \xrightarrow{suspend} ListentoCD$: An incoming call interrupts the listening to a CD. There are also two possible situations, in which an incoming call can interrupt the CD depending on the last active task:

1. Insert Disc $\rightarrow$ Play $\rightarrow$ Handle Incoming Call
2. Insert Disc $\rightarrow$ Play $\rightarrow$ Pause $\rightarrow$ Handle Incoming Call

Direct feature interactions are described by temporal task dependencies. By contrast, unintentional feature interactions are non-specified interactions between two features. Hence, they cannot be described explicitly in a task model. In an AIS, it is possible that feature interferences may result from indirect interactions between any features because all features use the same resources such as the GUI, the communication bus and often the same input and output devices. Especially situations in which different features are executed interleaved are error prone. These situations are described implicitly by a task model: all tasks in different features that have no temporal dependencies can be executed interleaved. In the example in Figure 4.19 this is the case for the feature Adjust Seat that has no temporal dependencies to the other features (nor vice versa). Therefore, there are multiple scenarios where the execution of Move Seat interferes with other tasks:

1. Insert Disc → Play → Start Moving → Stop Moving → Stop
2. Insert Disc → Play → Start Moving → Stop → Stop Moving
3. Insert Disc → Play → Pause → Start Moving → Stop Moving
4. ...

To conclude, this section showed how a *TTask* model describes intentional feature interactions by temporal task dependencies. This section also showed that a *TTask* model describes potential error prone unintentional feature interaction scenarios as well: namely when independent tasks are executed interleaved. The challenge is to generate task sequences that cover both intentional feature interaction scenarios as well as unintentional feature interaction scenarios.

4.6 Summary

Test case generation for feature interaction testing requires an appropriate test model. This test model should describe critical feature interaction scenarios. In this chapter we have shown that task models fulfill these requirements. A task is a feature-specific, goal driven interaction between system and environment. We can describe the black-box behavior of a feature by describing the set of tasks that are enabled by the feature. Task models describe tasks and their temporal dependencies. Hence, feature-specific scenarios are described by the temporal dependencies between a feature's tasks. Furthermore, intentional feature interactions are described by temporal dependencies between different features' tasks. Potential unintentional feature interactions can be inferred from a task model by deriving independent tasks that can be performed in an interleaved fashion.

We have given an overview of different techniques to describe feature interactions and task models with respect to test case generation. The existing approaches either lack the necessary formalization for test case generation or they are based on formal methods that require a deep understanding of the underlying concepts, which is usually not the case for testers in the automotive domain. In order to enable the specification of complex systems such as AIS and to enable the generation of test cases, we have introduced the new task modeling language *TTask*. *TTask* combines an easy graphical syntax with a formal foundation that enables the automated generation of test cases. A *TTask* model implicitly describes the space of all possible task sequences where each sequence is a potential test case. In the next chapter we introduce test selection criteria to select "interesting" task sequences that cover critical interaction scenarios.

5

Generating Task Sequences

This chapter covers the generation of task sequences from *TTask* models. A task model implicitly describes the space of all possible sequences of task executions. The number of different task sequences is potentially infinite. In order to use a task model for testing, we have to select from the space of possible task sequences the most “interesting” ones, namely the ones that are most likely to detect faults. In this chapter we introduce different test selection criteria for task models that focus on feature interaction scenarios. Furthermore, we describe test case generation from *TTask* models based on these criteria using model checking.

The contributions of this chapter are:

- The transformation of *TTask* models into *Promela* models, which is the input language of the model checker SPIN.
- New test selection criteria to cover feature interaction scenarios.
- Implementation of the introduced test selection criteria with linear temporal logic (LTL).

This chapter is structured as follows. First we give an introduction into related work of test case generation with model checkers. Next we describe the implementation of *TTask* models in *Promela*. Lastly for the remainder of this chapter, we introduce new test selection criteria for *TTask*.

5.1 Related Work

To generate test cases from a *TTask* model, we took the approach of utilizing a model checker to perform the generation. Model checking is a widely adopted technique for test case generation [RH01, BR04, HGW04, LP04, EKRV06, Cho07, HCL$^+$03, ZML07]. The basic idea is to treat test case generation as a reachability problem [HdMR05, EKRV06]. Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [CGP99].

The starting point is a behavior model of the system under test (SUT) that is defined in the input language of the targeted model checker. Test selection criteria are expressed as temporal properties based on this behavior model. These temporal properties are expressed in, for example, LTL, which is a widely-accepted and very expressive logic that can specify safety, fairness, and liveness properties. An LTL formula can be used to specify a property that must hold on all the execution paths of the model checker's input model, and such paths may be infinite both in number and in length [CGP99, TSL04]. Hence, LTL formulas are an appropriate means to define so called *trap properties*. For example, such a trap property might claim that a certain state or transition is never reached. When the behavior model is checked against the trap property, the model checker returns a counter-example that illustrates how the trap property is violated. For example, it shows how the state or transition described by the trap property is reached. This counter-example can be instantiated into a test case that can be used to test whether the SUT's behavior corresponds to the one specified in the counter-example.

Tan et al. use this approach in [TSL04] to generate test cases from requirements. They specify requirements in the form of LTL formulas and use these to generate test cases. Furthermore, they mutate these LTL formulas in order to increase the test coverage of the generated test suite. However, the effort of manually specifying LTL properties is high for large systems. Desirable is the automated generation of LTL formulas that cover specific system properties.

Rayadurgam and Heimdahl present in [RH01, HRV^{+}03] their approach of coverage-based test case generation with model checking. Their approach generates LTL formulas that cover structural properties of a system model that is given in the form of a transition system. The approach supports structural coverage criteria, such as transition coverage, guard coverage, and modified condition/decision coverage (MC/DC). Later, Heimdahl et al. presented in [HGW04] an evaluation of their structural coverage criteria. They performed an experiment that compared test suites generated by state, transition, and decision coverage with random generated test suites. They found out that the structural tests uniformly performed worse than randomly generated tests. They conclude that coverage criteria used in specification testing and specification based testing must be refined to better fit the semantics of the specification languages and the structure of the models captured in these languages. This corresponds to our approach of test case generation where we use a modeling language that is dedicated to modeling features and their interactions. Hence, we are able to define coverage criteria that explicitly cover feature interaction scenarios because these are explicitly included in the semantics of our modeling language.

5.2 Translating *TTask* into *Promela*

Model checking is a promising technique for test case generation. The main advantage is that one can use the advanced algorithms that are already implemented in a model checker. Hence, the effort shifts from the implementation of a test case generator to finding the best test selection criteria. Another advantage is that model checkers enable the simulation of their input models which provides the ability to validate the test model by simulation. Furthermore, it is possible to verify the correctness of the test model with respect to requirements that are specified in the form of LTL formulas.

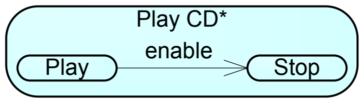
There are many different model checkers, each with specific advantages and disadvantages. In the following we describe the implementation of *TTask* models with *Promela*, the input language of the model checker SPIN [Hol97]. We use SPIN for three main reasons:

- **Mature tool support:** The model checker SPIN, which is based on *Promela*, has first been introduced 1995 and has been continuously developed since then.
- **Support for simulation and verification:** SPIN supports the verification of closed models, which is required for test case generation. Furthermore, it supports the simulation of open models, which we use to simulate *TTask* models. Using SPIN for both simulation and verification guarantees that both obey the same semantics.
- **Easy transformation of *TTask* to *Promela*** The semantics of *TTask* can be easily expressed with *Promela*, which enable a straightforward implementation of *TTask* in *Promela*.

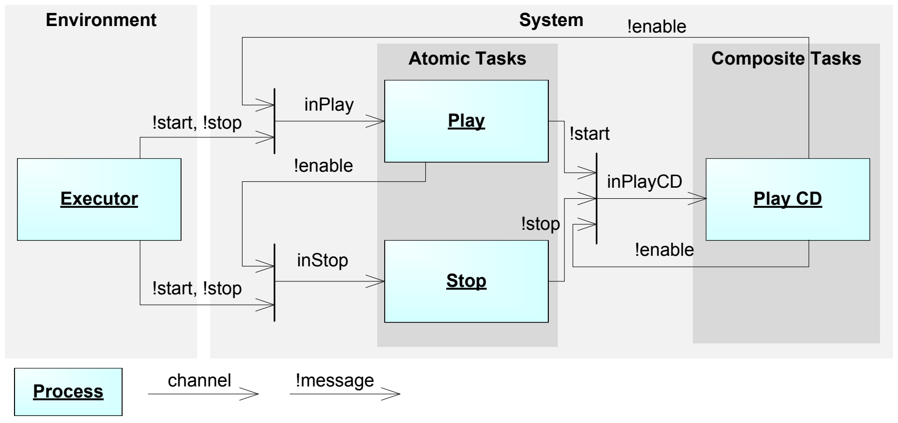
A *Promela* model is constructed from three basic types of object: processes, data objects and message channels. Processes communicate via message channels and share data via global data objects. *Promela* supports asynchronous and synchronous (called rendezvous in *Promela*) communication. A comprehensive introduction into the syntax and semantics of *Promela* can be found in [Hol97].

The mapping of *TTask* to *Promela* is a straightforward implementation of the semantics introduced in the previous chapter. The task model state is implemented by a set of global data objects that store the current mode for each task. Task specific mode changes are implemented in separate processes in *Promela*. Each of these processes have an input channel through which it receives events that trigger its mode change. Task dependencies are realized by messages that are sent between the task-specific processes. For example, the Play process sends *enable* to the Stop process. Figure 5.1 shows the mapping of a *TTask* model to the corresponding *Promela* model. Figure 5.1(a) shows a task model for a simplified CD player and Figure 5.1(b) shows the corresponding processes with the message channels and the messages that are sent between the processes. Figure 5.1(b) also shows the executor as a separate process that is responsible for starting and stopping *atomic tasks*. The *composite tasks* are started and stopped by their subtasks. When a task is started or stopped, it changes its mode and sends messages to dependent tasks corresponding to their temporal dependencies.

Listing 5.1 shows the initialization of a *TTask* model in *Promela*. Firstly, the different task modes and task signals are initialized using Mtypes. These enable the definition of symbolic names for numeric constants. Secondly, the message channels are initialized: each task has a separate incoming message channel. Finally the data objects that track the task states are initialized. The



(a) *TTask.*



(b) *Promela.*

Fig. 5.1. Mapping between *TTask* and *Promela*.

task modes are initialized with the *initial task mode* as defined in the task model.

```
// define all modes and signals
mtype = { Enabled, Disabled, Active, Suspended, SuspendedEnabled,
    DisabledSuspended, Resume, Suspend, Enable, Disable, Start, Stop};
// rendezvous channel
chan inStop   = [0] of {mtype};
chan inPlay   = [0] of {mtype};
chan inCDPlay = [0] of {mtype};

// set initial task modes
mtype StopMode = Disabled;
mtype PlayMode = Enabled;
mtype CDPlayMode = Enabled;
```

Listing 5.1. Task mode and message channel initialization in *Promela*.

In order to perform a verification in SPIN, the *Promela* model must be closed which means that the model does not depend on any external inputs. To accomplish this, we have to encapsulate all external inputs in a separate process. External inputs are the *start* and *stop* signals for *atomic tasks*. This process corresponds to the Executor in Figure 5.1(b). The implementation of the Executor is shown in Listing 5.2. The Executor process is the only process that is always active. The process loops infinitely, starting and stopping *enabled* tasks non-deterministically. If a task's mode is *enabled*, the Executor starts the corresponding task process and sends the *start* signal. When the message is sent, the Executor waits until all task mode changes have been performed. This is realized by the timeout statement that evaluates to true if and only if there are no executable statements in any of the concurrently running processes. This is true, when all task mode changes have been performed. After the timeout statement evaluates to true, the executor stops the started tasks and waits until the model is ready to start the next task. SPIN supports the simulation of *Promela* models as well. In this case the executor is replaced by the user who can manually start and stop tasks.

```
active proctype Executor()
{
  // non deterministic loop
  do
    :: PlayMode == Enabled ->
      // start Play
      run Play();
      inPlay!Start;
      // wait until Play is finished
      timeout;
      // stop Play
      run Play();
      inPlay!Stop;
      // wait until Play is finished
      timeout;
    :: StopMode == Enabled ->
      // start Stop
      ...
    :: else -> skip;
  od;
}
```

Listing 5.2. *Promela* process for the Environment.

Listing 5.3 shows the implementation of a task in *Promela*. When a task is started, it waits for a new message to arrive via its message channel. Depending on the incoming signal (*start*, *stop*, *enable*,...) and the current task mode, the process changes its corresponding task's mode. Then it starts the processes for its subtasks and its related tasks and sends signals depending on the incoming signal. For example, when a process receives a start signal, it subsequently sends a disable signal to all processes that implement tasks that are related by the *disable* dependency. After all task processes are notified, the task process ends.

The execution order of statements in different processes in SPIN is not deterministic due to SPIN's interleaving semantic. However, in our *TTask* implementation in *Promela* there is always only one process active. Therefore we avoid the interleaving semantics in order to restrict the number of possible paths in the *Promela* model. This is achieved by using the rendezvous communication between all processes in combination with the definition of atomic sequences that are non-interleaved with other processes. Atomic sequences begin with the atomic statement and are surrounded by curly braces.

```
proctype Play()
{
  atomic{
    mtype new;
    // wait for new message
  inPlay?new −>
  if
    :: new == Start && PlayMode == Enabled −>
      // reveiced start
      if
        // if the parent is not active send start
        :: PlayCD != Active −>
          run PlayCD();
          PlayCD!Start;
        :: else −> skip;
      fi;
      // set task active
      PlayMode = Active;
    // received stop
    :: new == Stop && PlayMode == Active −>
      // set task disabled
      PlayMode = Disabled;
      // run Stop task process
      run Stop();
      // send enable to the Stop task
      inStop!Enable;
    // receive enable
    ...
    :: else −> skip;
  fi;
 }
}
```

Listing 5.3. *Promela* process for the Task Play.

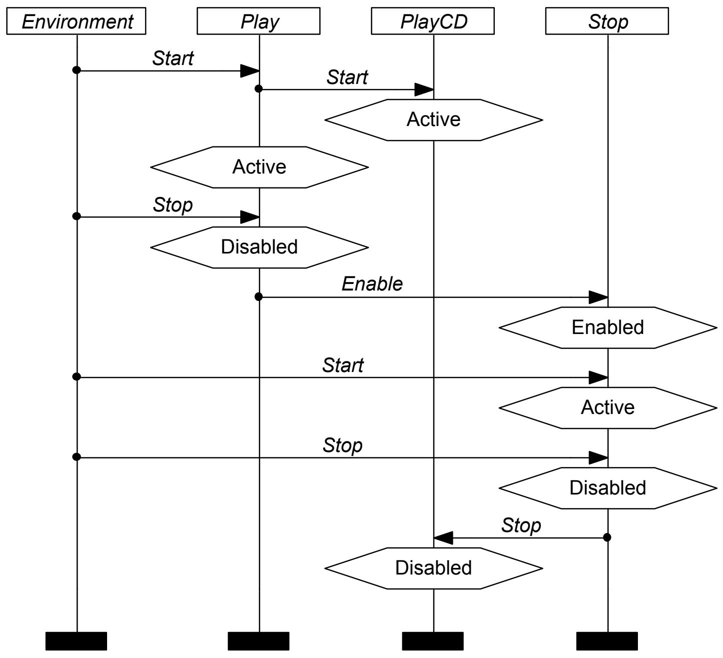


Fig. 5.2. Execution of the tasks Play and Stop.

Figure 5.2 shows an MSC that describes the mode changes that take place when the Executor performs the tasks Play and Stop from the task model in Figure 5.1(a). The MSC is automatically generated from a SPIN trail. The diamond shapes in the MSC show the current task mode. In the MSC, first the task Play is started by the Executor. When it is started, Play sends *start* to its parent task PlayCD, which changes its mode to *active*. The task Play is stopped and enables the task Stop which is then performed.

Based on the introduced transformation of *TTask* models into *Promela* models, we are able to use the model checker SPIN for test case generation. The complete translation from *TTask* to *Promela* can be found in Appendix A.1.

5.3 Task Sequence Generation

This section introduces the generation of task sequences using model checking. We demonstrate this using the task model in Figure 5.3 which has already been introduced in Chapter 4.5.

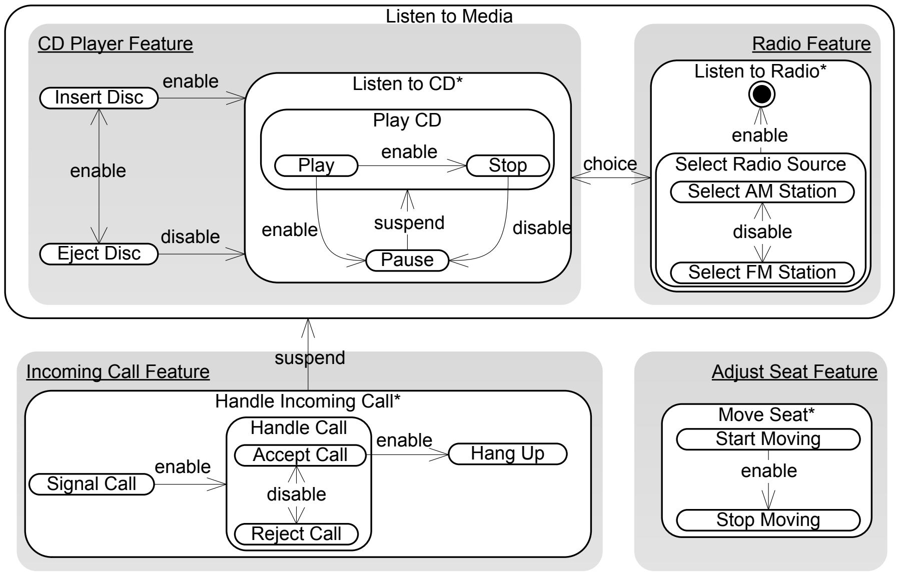


Fig. 5.3. Example task model.

In order to select specific task sequences, we define trap properties that the model checker tries to fulfill. For example, a trap property might claim that the task Play is never active. The model checker explores the state space and tries to find a state where the property is violated. If a property violation is detected, namely the task Play is active, the model checker produces a counter-example by giving a trail that leads to the violation. The trail is a task sequence consisting of task executions and the resulting task model states.

The model checker SPIN supports two ways of specifying trap properties: LTL formulas and *Promela* code. In this section, we focus on the specification of trap properties with LTL. A simple trap property for the task model in Figure 5.3 that claims that the task Signal Call is never active is defined as follows:

$$\neg\Diamond(SignalCall = active)$$

The *eventually operator* $\Diamond\phi$ defines that a given property ϕ has to hold eventually on the subsequent path. During the execution of a *TTask* model, the subsequent path corresponds to the subsequent task model states. Using this operator it is possible to define that for all sequences in a task model Signal Call is never active. The predicate $SignalCall = active$ specifies a value of the global data object that stores the current mode of the task Signal Call.

Furthermore, LTL supports the until operator $\phi\ U\ \tau$: a predicate τ must hold at the current or a future position in the path, and ϕ has to hold until that position. At that position ϕ needs not to hold any more.

SPIN finds a violation of this claim for the task model in Figure 5.3, namely the sequential execution of the tasks Insert Disc and Play. The task sequence for the resulting counter-example is depicted in Figure 5.4. The task sequence consists of the task model's initial state S_0 and the task model states S_1, S_2, S_3 that result from starting and stopping the atomic tasks Insert Disc and Play. Each task model state comprises the current mode of all tasks in the task model. This enables selecting arbitrary task sequences by specifying sequences of task modes.

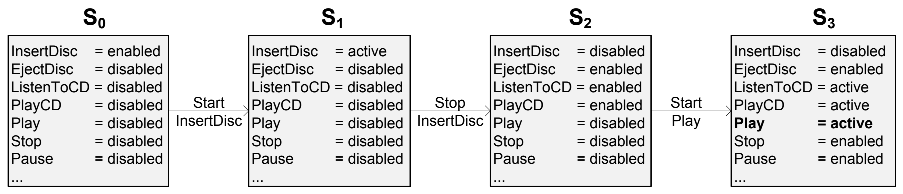


Fig. 5.4. Task sequence where $\Diamond(Play = active)$.

5.4 Task-based Test Selection Criteria

Tasks can be executed multiple times, which results in a potentially infinite set of task sequences. Thus it is usually not possible to test all task sequences. Rather we want to select the most "interesting" task sequences from the space of all task sequences. The study in Section 3.2 showed that feature interactions are error prone. Hence, task sequences that involve feature interaction are "interesting" because they are likely to be error prone. This section introduces test selection criteria for task models that select such "interesting" task sequences.

The process of task-based test case generation is summarized in Figure 5.5. First, a *TTask* diagram is compiled into its *Promela* representation. In the next step a trap property generator derives a set of LTL formulas from the *TTask* diagram and a given test selection criterion. The generated *Promela* code and LTL formula are input to the model checker SPIN which tries to generate a counter-example in the form of a task sequence for each LTL formula. The resulting task sequences are test cases that we can use for testing.

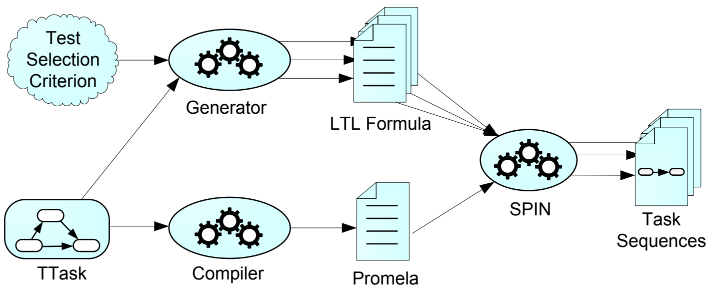


Fig. 5.5. The task sequence generation process.

5.4.1 Explicit Selection Criteria

Explicit selection criteria specify conditions that must be satisfied by one task sequence. During testing often specific sequences must be selected that are especially error prone or that are not covered by other criteria. The test engineer creates a test case specification in LTL that selects one “interesting” task sequence. The main advantage is that explicit selection criteria give precise control over the generated sequences. This can be used to select error prone sequences that are not covered by coverage-based criteria and to increase the coverage for specific tasks. Another use case is the generation of *test preambles*. A test preamble is a sequence of inputs after which the system is in a specific state. A test preamble is necessary to test specific system properties. For example, a test engineer wants to test if the GUI shows the correct icon when the CD player is paused. The engineer selects a task sequence in which the task Pause is active. After the task sequence is executed the CD player is paused and the actual test for the correct icon can be executed.

In the following we introduce different examples of the exclusive selection of task sequences:

Single Task Execution. When a task changes its state from *active* to *disabled*, it is performed. For example, the following LTL formula selects a task sequence where the task Signal Call is performed:

$$\neg\Diamond(\text{SignalCall} = \text{active} \wedge \Diamond\text{SignalCall} = \text{disabled})$$

Multiple Task Execution. We can extend the previous LTL formula to support arbitrary sequences of task executions $t_1, ..., t_n$:

$$\neg\Diamond(t_1 = \textit{active} \wedge \Diamond(t_1 = \textit{disabled} \wedge \Diamond(t_2 = \textit{active} \wedge \Diamond(t_2 = \textit{disabled} \wedge ...))))$$

Arbitrary Task Model State Selection. Furthermore, arbitrary task model states can be specified that must be fulfilled by a task sequence. For example, the condition that the seat is adjusted while the CD Player is suspended by an incoming call is defined as follows:

$$\begin{aligned}\neg\Diamond(&ListenCD = suspended \wedge HandleIncCall = active\\ &\wedge MoveSeat = active \wedge ListenRadio = suspended\_disabled)\end{aligned} \tag{5.1}$$

The suspension of the CD Player by the incoming call is implicitly defined based upon the *suspend* dependency between the tasks. Hence, a task model state, in which the task Handle Incoming Call is active and the task Listen to CD is suspended, implies that the incoming call suspended the CD Player. This is possible because we rule out that the CD Player is suspended by the radio. The main disadvantage of explicit selection criteria is that the LTL formula must be maintained for each test case separately. Nevertheless, the explicit selection of test cases is often necessary as an addition to coverage-based test selection criteria.

5.4.2 Coverage-based Selection

Coverage-based test selection criteria specify conditions that must be satisfied by a set of task sequences. They represent certain aspects of a system that the engineer wants to test. A coverage-based test selection criterion derives a set of properties from a given task model. These properties are trap properties that are expressed as LTL formulas. In the following we introduce several coverage criteria for task models.

Task Coverage Criterion. This test selection criterion ensures that each task is performed at least once. This is achieved by generating a set of task sequences from a given *TTask* model, where each task is performed at least once in one of the task sequences. Although, this criterion seems fairly trivial, it covers feature interactions. A task sequence consists of a set of task model states, and each task model state comprises the current modes of all tasks. Hence, when an active task disables another task the corresponding mode in the task model state is *disabled*. This information can be used to define test oracles that test if the corresponding task is really disabled in the SUT. As a consequence, the task coverage criterion covers disabling dependencies as well. However, the task coverage criterion covers only one of the possible disabling scenarios.

For a given task model $M = (T, S_0, \Delta, \Phi)$ the *task coverage criterion* creates for each task $t_i \in T$ an LTL formula that selects a task sequence $\mathrm{TS}_i = \langle S_0, ..., S_n\rangle$, where $S_i = \langle m_0, ..., m_n\rangle$ in which t_i is performed:

$$\neg\Diamond((t_i = active) \wedge \Diamond(t_i = disabled))$$

Then SPIN is invoked for each of these generated LTL formulas in order to generate a counter-example.

Interruption Coverage Criterion. The *interruption coverage criterion* explicitly covers all feature interaction scenarios in which one task suspends

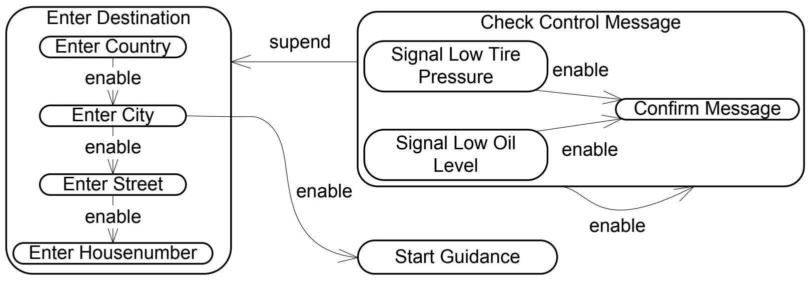


Fig. 5.6. Interruption coverage example.

another *active* task. A task can be suspended in different situations. For example, the task model in Figure 5.6 describes two tasks, Enter Destination and Check Control Message. Check control messages can interrupt the destination input, which is modeled by the *suspend* dependency. The interruption can occur after Enter Country, Enter City, Enter Street have been performed. Furthermore, Enter Destination is interrupted when the task Signal Low Oil Level or the task Signal Low Tire Pressure is performed. Hence, there are eight different interruption scenarios that must be selected by the *interruption coverage criterion.*

First, the *interruption coverage criterion* generates a set of task triples P that describe a potential interruption scenario:

$$P = \{\langle t_i, t_j, t_p \rangle \mid t_i, t_j, t_p \in T\}$$

Such a triple $\langle t_i, t_j, t_p \rangle \in P$ describes a potential interruption scenario if t_i and t_j are atomic tasks and if there exists a *suspend* dependency from t_i or one of its ancestors to a task t_p which is an ancestor of the task t_j:

$$\begin{aligned} \Phi(t_i) = \emptyset \ \wedge\ \Phi(t_j) = \emptyset \ \ \wedge\ \exists \delta \in \Delta.\ \delta = \langle t_x, t_p, suspend \rangle \\ \wedge\ t_x \in closure(t_i) \ \wedge\ t_p \in closure(t_j) \end{aligned} \tag{5.2}$$

In Figure 5.6, example task triples are:

- $\langle SignalLowTirePressure, EnterCountry, EnterDestination \rangle$
- $\langle SignalLowTirePressure, EnterCity, EnterDestination \rangle$
- $\langle SignalLowTirePressure, EnterStreet, EnterDestination \rangle$
- $\langle SignalLowOilLevel, EnterCountry, EnterDestination \rangle$
- ...

In the next step, the criterion creates for each task triple $\langle t_i, t_j, t_p \rangle \in P$ the following LTL formula:

$$\begin{aligned} \neg\Diamond(t_j \neq disabled \ \wedge\ (t_j = active\, U\ (t_j = disabled \ \wedge\ t_i \neq active \\ U\ (t_i = active \ \wedge\ t_i \neq disabled \\ U\ (t_i = disabled \ \wedge\ t_p \neq active\, U\ t_p = active))))) \end{aligned} \tag{5.3}$$

Finally, SPIN is invoked for each LTL formula in order to generate a counter-example. The result is the set of counter-examples that were found by SPIN for the given set of LTL formulas.

The *interruption coverage criterion* selects for each *suspend* dependency a corresponding task sequence in a task model. The number of generated task sequences can be restricted by focusing only on feature interactions. This is achieved by restricting the set of task pairs to the ones that involve a feature interaction. Hence, for a given a set of features $F = f_1, ..., f_n$, a task pair $\langle t_i, t_j \rangle \in P$ must additionally hold that:

$$\exists f_x \in F.\, t_i \in f_x \ \wedge \ t_j \notin f_x$$

Enabling Coverage Criterion. The *enabling coverage criterion* covers all task sequences where a task enables another task. Such an enabling situation is covered by a task sequence in which an enabling task and one of its enabled tasks are performed subsequently. In order to cover all enabling scenarios of the task Enter City in Figure 5.6 we have to create two different task sequences:

1. ...→ Enter City → Start Guidance → ...
2. ...→ Enter City → Enter Street → ...

First, the *enabling coverage criterion* generates a set of atomic task pairs

$$P = \{\langle t_i, t_j \rangle \mid t_i, t_j \in T \wedge \Phi(t_i) = \emptyset \ \wedge \ \Phi(t_j) = \emptyset\}$$

which describe a potential enabling scenario. A task pair $\langle t_i, t_j \rangle$ describes a potential enabling scenario when it holds[1]:

$$\exists t_p \in T.\, t_p \in closure(t_i) \ \wedge \ t_j \in enabledBy(t_p) \tag{5.4}$$

In the next step, the criterion creates for each task pair $\langle t_i, t_j \rangle \in P$ the following LTL formula:

$$\neg\Diamond(t_i = active \ \wedge \ t_j = disabled \wedge \\ \Diamond((t_j = active \ \wedge \ t_i = disabled) \ \wedge \ \Diamond(t_j = disabled))) \tag{5.5}$$

In order to select only enabled dependencies between different features, the extension of the enable criterion is similar to the interruption coverage criterion.

Concurrency Coverage. The *concurrency coverage criterion* covers all task sequences where two independent tasks are executed in an interleaved fashion. Interleaving of tasks is only possible if they have no temporal dependencies. Thereby, the *concurrency coverage criterion* covers task sequences where a potential unintentional feature interaction may occur. Figure 5.7 shows the two independent tasks Check Control Message and Move Seat. There are multiple task sequences in which these two tasks are executed concurrently. These tasks are performed concurrently by executing their subtasks in an interleaved fashion. The subtasks of Check Control Message each can be performed directly

[1] for the definition of enabledBy(t) see definition 4.12 in Chapter 4.3

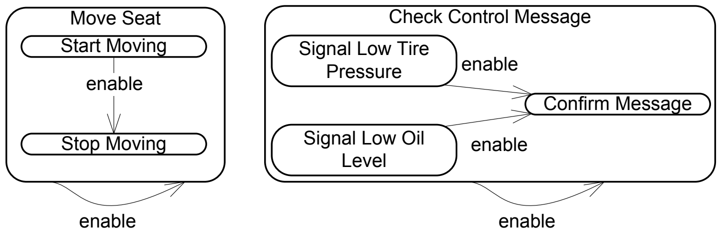


Fig. 5.7. Concurrency coverage example.

after Start Moving or Stop Moving and vice versa, which results in a total of six different task sequences.

First, the *concurrency coverage criterion* generates a set of atomic task pairs

$$P = \{\langle t_i, t_j \rangle \mid t_i, t_j \in T \wedge \Phi(t_i) = \emptyset \;\wedge\; \Phi(t_j) = \emptyset\}$$

which describe a potential interleaved execution of two independent tasks. The tasks t_i and t_j are independent if they hold[2]:

$$\begin{aligned} &\nexists t_p \in T.\, t_p \in closure(t_i) \\ &\qquad \wedge\; t_j \in (suspendedBy(t_p) \;\cup\; enabledBy(t_p) \;\cup\; disabledBy(t_p)) \end{aligned} \tag{5.6}$$

In the next step, the criterion creates for each task pair $\langle t_i, t_j \rangle \in P$ the following LTL formula:

$$\begin{aligned} &\neg\Diamond(t_1 = active \;\wedge\; t_2 = disabled \wedge \\ &\qquad\qquad \Diamond((t_2 = active \;\wedge\; t_1 = disabled) \;\wedge\; \Diamond(t_2 = disabled))) \end{aligned} \tag{5.7}$$

In order to select only independent tasks from different features, the extension of the concurrency criterion is similar to the interruption coverage criterion.

5.4.3 Random Selection

The random selection of test cases is no test selection criterion in the classical sense. Nevertheless, the random selection of test cases is a useful technique in addition to classical coverage criteria. The random generation of inputs often generates unusual sequences that a human tester may not consider. Furthermore, it enables the generation of large test cases that exercise a system for a longer period of time. SPIN supports the random simulation of a given *Promela* model. Using this feature it is possible to generate random task sequences from a given *TTask* model without additional effort.

5.5 Summary

Task models describe feature interactions and hence enable the generation of test cases that cover critical interaction scenarios. Using a model checker

[2] for the definition of $disabledBy(t)$ see definition 4.5 in Chapter 4.3

for test case generation takes advantage of the advanced search algorithms of a model checker. Test case generation, therefore, is reduced to defining trap properties that describe scenarios that are potentially error prone. The model checker automatically generates counter-examples for these trap properties where each counter-example is a potential test case. In this chapter, we discussed the process of test case generation from *TTask* models. First we described the automated transformation of a *TTask* model into a *Promela* model. Furthermore, we introduced multiple test selection criteria that select task sequences that systematically cover feature interaction scenarios. Although, we introduced concrete criteria for test case selection we do not claim that the introduced criteria are complete. Rather, model checking based test case generation enables the definition of custom test selection criteria that are adapted to the respective system or domain.

Part III

Test Case Instantiation

6

Refining Task Sequences

The previous chapter introduced test selection criteria that enable the generation of "interesting" task sequences. Based on these test selection criteria it is possible to generate task sequences that systematically cover feature interaction scenarios. However, task sequences lack two elementary parts of a test case: they describe no input and no output behavior. Hence, in order to use task sequences for testing, they must be enriched with the missing behavior. This chapter presents two approaches that enable the refinement of task sequences based on existing behavior models in order to transform task sequences into test cases. The contributions of this chapter are:

- An approach for task sequence refinement based on scenarios.
- An approach for task sequence refinement based on state machines.

This chapter is structured as follows: first we describe the abstraction gap between task sequences and the SUT; next we give an overview on related work dealing with the refinement of test models; and in the rest of this chapter we introduce two way of using existing models to enrich task sequences with the missing input and output behavior.

6.1 Testing with Task Sequences

Task sequences represent interaction scenarios between system and environment. The test selection criteria that where introduced in the last section enable the generation of task sequences that cover feature interaction scenarios where faults are likely to occur. Hence, each task sequence represents a scenario that must be tested in order to ensure that no feature interaction results in a feature interference.

A task sequence describes a sequence of task executions and the corresponding task modes. Figure 6.1 shows a simple task model that describes the tasks Incoming Call and Listen to Radio. One "interesting" task sequence is the one where the incoming call suspends the radio. Figure 6.2 shows the corresponding task sequence.

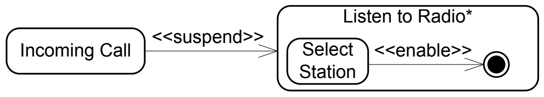


Fig. 6.1. Task model.

A tester can now perform this task sequence and test the system for correct behavior by executing the following steps:

1. Start the radio by selecting a station.
2. Start an incoming call.
3. Test if the incoming call is signaled.
4. Test if the radio is suspended.

The tester executes a task sequence by implicitly mapping the tasks to the corresponding environment inputs and the task modes to the corresponding system states. For example, the task Select Station is mapped to the IDrive controller inputs which are necessary to select a station and the task mode *suspended* from the task Listen to Radio is mapped to testing whether the radio is suspended. The tester performs this mapping based on the knowledge of the system and the system specification.

However, the effort of manual test execution is high, in particular when test cases must be repeated multiple times due to different system configurations or due to system changes. Furthermore, test case generation potentially produces a large number of test cases. For example, the *interleaving coverage* criterion produces a task sequence for each interleaved execution of two independent tasks. Thus the automated execution of test cases is desirable in order to reduce testing effort and to enable a continuous testing process. This requires executable test scripts that automatically trigger and observe the system. Therefore it would be desirable for a test script to automatically perform the steps of a manual tester. Listing 6.1 shows such a Python test script for the

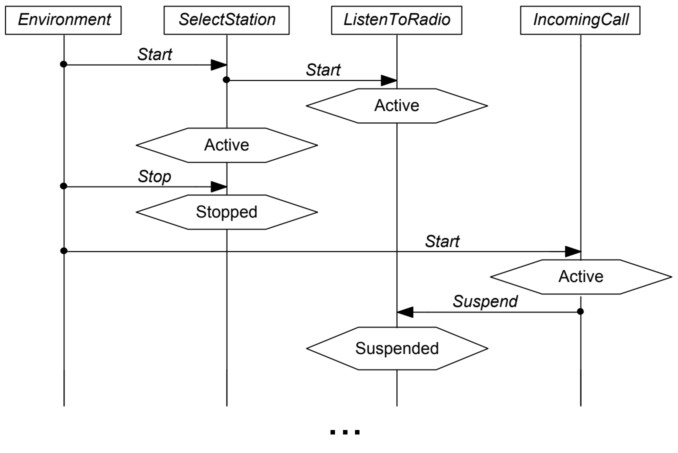


Fig. 6.2. Task Sequence: Incoming Call suspends Listen to Radio

task sequence in Figure 6.2. The test script can be executed automatically and perform the same tests as the manual tester.

```
# Navigate to Radio Station menu
ZBE.Right(1)
ZBE.Right(1)
...
# Start radio by selecting a station
ZBE.Press(1)
# Start incoming call
MobilePhone.DialNumber("0179xxxxxxxx")
#Test: incoming call signaled
if (MMI.getText() != "Incoming Call"):
  Log.error("Incoming Call not signaled")
#Test: radio suspended
if (Audio.getChannel() != Channel.Phone):
  Log.error("Wrong audio channel")
```

Listing 6.1. Test script.

In order to automatically transform a task sequence into such a test script, the tester's implicit mappings of tasks to test triggers and system states must be made explicit. In this chapter and the following one we describe how this can be accomplished.

The transformation of an abstract task sequence into an executable test script is performed in two steps:

1. **Refinement:** A task sequence is refined by enriching a task with the corresponding test inputs and system outputs. For example, the task Select Radio Station is refined by adding the IDrive controller inputs that are required to select a station. The refinement of task sequences is described in the remainder of this chapter.
2. **Test Script Generation:** The transformation of a refined task sequence into the test script language. For example, by transforming a task sequence into a Python script. The transformation of abstract test cases into executable test scripts is the topic of the next chapter.

6.2 Related Work

The concept of mapping abstract models to more concrete ones is a common concept to bridge different abstraction levels. There are several publications that use this concept to refine generated test cases.

The work presented by Pfaller et al. is based on the formalization of user requirements in the form of services [PFH$^+$06]. They model the refinement of services into underlying functional models on lower abstraction levels. Based on these relations, they reduce the underlying functional models into the parts that are touched by the service. From these reduced functional models they generate service specific test cases. Their approach is not appropriate for instantiating test cases that are generated from task models because our goal is to generate test cases that cover multiple services. Therefore task sequences cover large parts of the system. Thus reducing the underlying functional models still results in large models with a potentially large number of test cases.

Fröhlich and Link map use cases to statecharts using preconditions and post-conditions [FL00]. Based on this relations test cases for each use case are semi-automatically generated from the statecharts using artificial intelligence (AI) planning. Their approach is similar to our approach of state-machine-based refinement that we introduce in the second part of this chapter. However, task models express different task modes resulting from task executions, for example, when a task is suspended by another task. Such temporal dependencies are not defined for use cases. In order to test these dependencies our approach is to include them during test case refinement.

6.3 Refining Task Models with Scenarios

The most intuitive approach to refine a task sequence is to describe for each task the corresponding test triggers and the corresponding system reactions in the form of a scenario. A scenario is a sequence of interactions between environment and system. Such a scenario is a system trace, which we introduced in the beginning of Chapter 4.1. The interaction is defined by a sequence of input actions and output actions. Let I be the set of all possible environment inputs and O be the set of all possible system outputs. Hence, a scenario C is a sequence of environment inputs and outputs:

$$C = \langle i_1, ..., i_n \rangle, \text{ where } i_i \in I \cup O$$

Our definition of a scenario is straightforward and does not include timing information and conditionals. More thorough definitions of scenarios are given in [BK98]. Nevertheless, our scenario definition is sufficient enough to describe the refinement of a task model. In Chapter 4.1, we used task models to abstract from concrete system traces by combining specific streams of actions into tasks. However, in order to refine task sequences we have to reverse this step: the basic idea is to map each atomic task to a corresponding scenario that describes the interaction between environment and system. Hence, composite tasks are represented by the composed scenarios of their subtasks.

A task model $M = (T, S_0, \Delta, \Phi)$ is refined by mapping each atomic task to a scenario $C = \langle i_1, ..., i_n \rangle$. Let A be the set of all atomic tasks

$$A = \{t \in T\}, \text{ where } \Phi(t) = \emptyset$$

and $SCENARIO$ be the universe of all scenarios. The task to scenario mapping is then defined by a relation α:

$$\alpha : A \rightarrow SCENARIO$$

Based on the task to scenario mapping we can define the refinement of a task sequence by a sequence of scenarios. Given a task sequence $TS = \langle S_1, ..., S_n \rangle$. Then a sequence of scenarios $SC = \langle C_1, ..., C_n \rangle$, where $C_i \in SCENARIO$, is a valid refinement of TS if it holds: [1]

$$\forall C_i \in CS.\ \exists S_{2i-1} \in TS.\ S_{2i-1}[t_j] = active \wedge \alpha(t_j) = C_i$$

The next section demonstrates the scenario-based refinement for a small example.

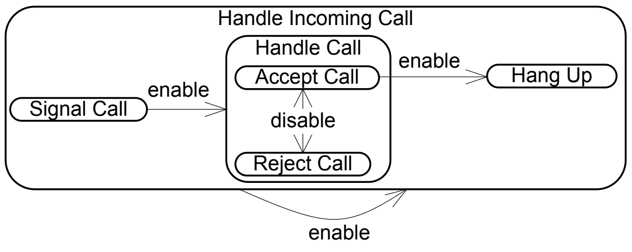


Fig. 6.3. Incoming call task.

6.3.1 Example for Scenario-based Refinement

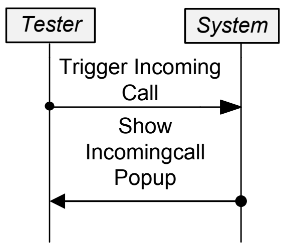
Figure 6.3 shows the task Handle Incoming Call that has been discussed earlier in Section 4.5. Each of the atomic tasks in Figure 6.3 are mapped to a corresponding scenario that describes the task-specific interaction between environment and system. The scenarios are shown in Figures 6.4(a) - 6.4(d).

When we create a task sequence from the task model, we can add the missing behavior by replacing each active task in the task sequence by its scenario description. Figure 6.5 shows the refined version of the task sequence Signal Call $\rightarrow$ Accept Call $\rightarrow$ Hang Up. The refinement of the task sequence results in a scenario that describes the sequential execution of the three task-specific scenarios.

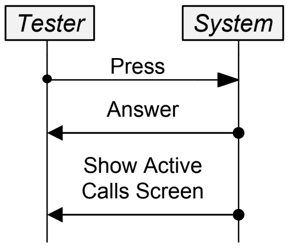
6.3.2 Test Case Composition with Task Models

In the previous section, a given task model is refined by creating scenarios for each atomic task. This represents a top-down approach where the task model is created before the scenarios are created that describe the concrete input and output behavior of atomic tasks. However, it is also possible to use task models in a bottom-up approach, where first scenarios or test cases are created. For example, in an early phase of integration testing, usually test cases are created to test specific use cases. These test cases can be performed

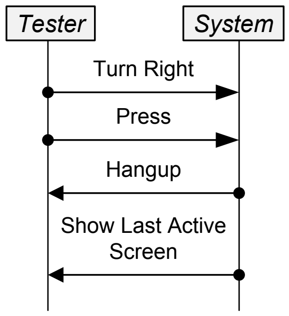
[1] A task sequence results from the sequential execution of atomic tasks: $start(t_i), stop(t_i), ...$ Hence, $S_1, S_3, S_5, ...$ are task model states where one atomic task is active and $S_0, S_2, S_4, ...$ are task model states where no atomic task is active. As a consequence, a scenario C_i corresponds to a task model state S_{2i-1} where the corresponding atomic task is active.



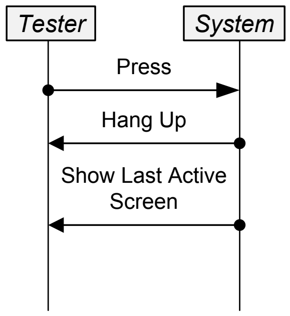
(a) Signal Call.



(b) Accept Call.



(c) Reject Call.



(d) Hang Up.

Fig. 6.4. Scenarios for the atomic tasks in Handle Incoming Call.

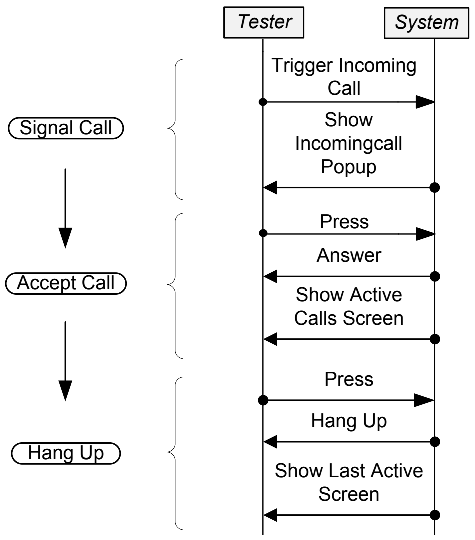


Fig. 6.5. Scenario for the task sequence Signal Call $\rightarrow$ Accept Call $\rightarrow$ Hangup.

sequentially in order to test more complex usage scenarios. However, an arbitrary combination of test cases is usually not possible. Rather, test cases can only be composed if a test's precondition matches the postcondition of the previously executed test. Task models can be used to describe these pre- and postconditions by using the *enable* operator. Even more, they provide the means to describe more complex execution scenarios by using the other temporal operators *disable* and *suspend*. Our approach is to create a task for each test case and describe the temporal dependencies to the other test case tasks. Based on such a task model, task sequences can be generated using the previously introduced test selection criteria which results in an increased number of tested usage scenarios. This enables the generation of more-sophisticated testing scenarios where multiple scenarios can be performed sequentially.

6.3.3 Limitations of Scenario-based Refinement

The scenario-based refinement of task sequences is a straightforward approach to refine task sequences by describing task-specific interaction between environment and system in the form of scenarios. However, the scenario-based refinement works only when the actual interaction between system and environment is independent of the history of executed tasks. For example, when multiple tasks are performed via a GUI, the currently focused element depends on the last task that has been performed before. This is particularly important for UIs that comprise of multiple screens which the user can navigate between, such as the GUI of an AIS.

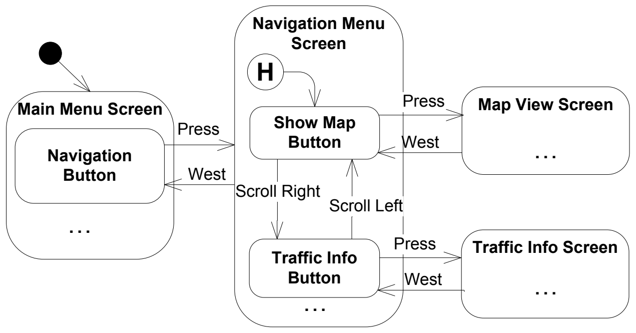


Fig. 6.6. GUI behavior and the corresponding task model.

Figure 6.6 shows an extract of the behavior of an automotive GUI. The behavior is specified by a statechart that describes the focus behavior of the navigation menu. The events represent inputs by the IDrive controller, for example, the event press is triggered by the user when the controller is pressed. The user can perform two tasks in the statechart: view the map or retrieve the actual traffic information. Figure 6.7 shows the corresponding tasks.

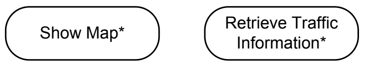
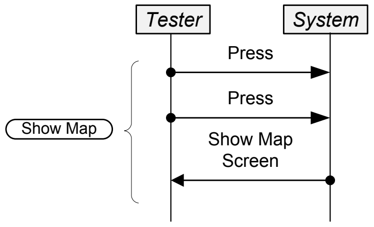
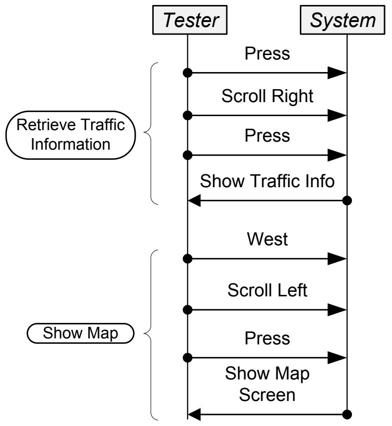


Fig. 6.7. Two independent tasks.

Both tasks can be performed in any order and therefore have no temporal dependencies. The scenarios for both tasks differ depending on the history of performed tasks. Figure 6.8(a) shows the scenario in which only the task Show Map is performed and Figure 6.8(b) shows the scenario in which Show Map is performed after the task Retrieve Traffic Info has been performed. The scenarios for the task Show Map differ, because in the second scenario other input events are necessary to reach the state Map View Screen.



(a) Show Map Scenario.



(b) Show Map Scenario after Retrieve Traffic Info.

Fig. 6.8. Two different scenarios for the same task Show Map.

As this example shows, there are situations in which tasks cannot be mapped to a single scenario. Thus, the scenario-based refinement cannot be applied in every situation. Nevertheless, scenario-based refinement is an easy way to refine task models if it is applicable. In the following section, we present our approach that enables the refinement of task sequences where scenario-based refinement cannot be applied.

6.4 Refining Task Models using State Machines

The previous section showed that scenario-based refinement cannot be applied under all circumstances. Therefore, we require a more general approach for the refinement of task sequences that comprises the history of previously executed tasks.

The idea behind our approach is to reuse existing behavior models that describe the concrete interaction between system and environment in order to refine task sequences. For example, a dialog model of the GUI describes the user inputs and the corresponding system reaction. This model contains the necessary information to generate the user inputs to perform a task. Furthermore, this can be achieved in a way that is independent of the history of task executions. We can use such a model to refine a task sequence by mapping a task model to existing software component or feature models. The mapping is performed by mapping a task to one or more states in software component models. Our approach is similar to the one presented by Fröhlich and Link in [FL00]. A task is mapped to a state machine by defining a task's precondition and postcondition in the state machine. For example, the goal of the task Signal Incoming Call would be the telephone component being in the state Incoming Call, the CD player component in the state Paused and the GUI in the state Incoming Call Screen, as depicted in Figure 6.9.

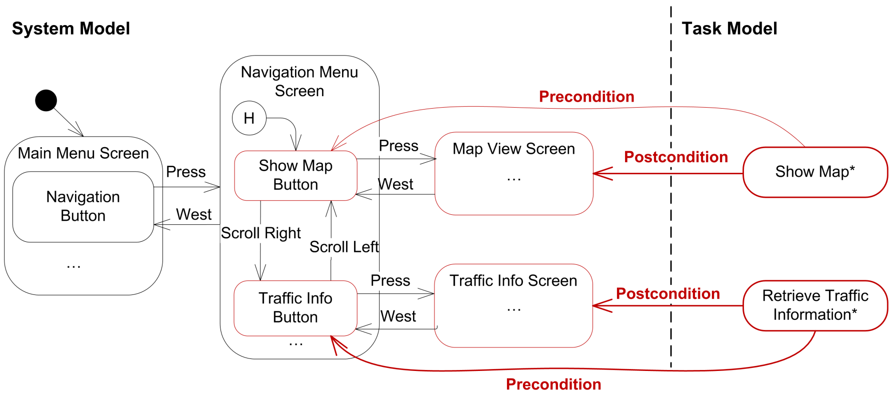


Fig. 6.9. Task to software component mapping.

With the information from the mapping it is possible to derive the input behavior and expected output behavior from the component models and therefore be able to generate task scenarios that include the history of task executions. Figure 6.10 shows the resulting process of combined test generation. A task model generates test cases for integration testing. Furthermore, the task model is mapped to existing component models. The generated task sequences are now refined into test cases by generating the missing input and output behavior from the component models.

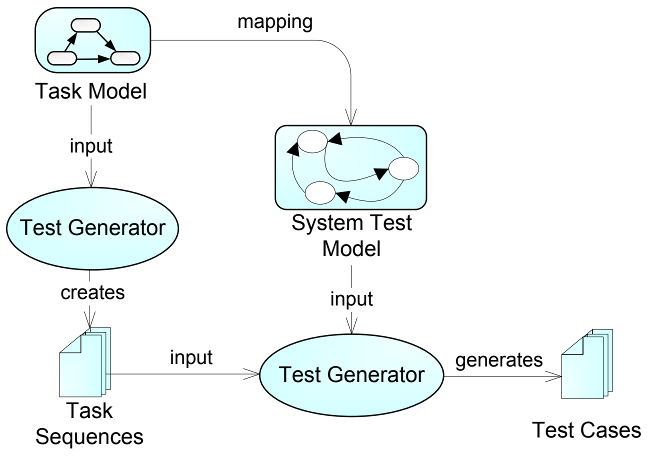


Fig. 6.10. Combined test case generation.

In the following we give a more formal definition of the mapping between a task model and a state machine. Given a task model $M = (T, S_0, \Delta, \Phi)$ and a finite state machine SM $= (\Sigma, S, s_0, \delta, F)$ where:

- Σ is the input alphabet (a finite, non-empty set of symbols).
- S is a finite, non-empty set of states.
- $s_0 \in S$ is the initial state.

- δ is the state transition function: $\delta : S \times \Sigma \rightarrow S$.
- F is the set of final states, a (possibly empty) subset of S.

The mapping between a task model $M = (T, S_0, \Delta, \Phi)$ and a state machine $\mathrm{SM} = (\Sigma, S, s_0, \delta, F)$ is defined by two relations *pre* and *post*:

$$pre : T \rightarrow S$$

$$post : T \rightarrow S$$

For example, the mapping for the task Show Map in Figure 6.9 is described by:

$$pre(ShowMap) = MapViewButton$$

$$post(ShowMap) = MapViewScreen$$

The mapping between a state machine and a task model enables the automated generation of the missing input behavior. An execution sequence E of a state machine SM has the form:

$$E = \langle s_0, i_1, s_1, ..., i_n, s_n \rangle, \text{ where } s_i \in S$$

where holds:

- s_0 is the initial state of S.
- $\forall i_j \in E.\ \exists s_{j-1} \in E.\ \exists s_j \in E.\ \delta(s_{j-1}, i_j) = s_j$

Given a task model $M = (T, S_0, \Delta, \Phi)$ and a state machine $\mathrm{SM} = (\Sigma, S, s_0, \delta, F)$. An execution sequence:

$$E = \langle s_0, i_1, s_1, ..., i_n, s_n \rangle$$

is a valid refinement of a task sequence:

$$TS = \langle S_0, ..., S_n \rangle, \text{ where } S_i = \langle m_0, ..., m_w \rangle$$

when for all pairs of atomic tasks that are performed consecutively:

$$\forall t_a \in T.\ \forall t_b \in T.\ \Phi(t_a) = \emptyset \wedge \Phi(t_b) = \emptyset \wedge \exists S_i \in TS.\ \exists S_{i+2} \in TS. \\ S_i[t_a] = active \wedge S_{i+2}[t_b] = active \quad (6.1)$$

holds:

$$\exists s_c, s_d, s_e, s_f \in E.\ pre(t_a) = s_c \ \wedge\ post(t_a) = s_d \wedge \\ pre(t_b) = s_e \ \wedge\ post(t_b) = s_f \ \wedge\ (c < d < e < f) \quad (6.2)$$

A task sequence can be refined by an additional test case generator that takes a task sequence as input and creates one or more valid paths in the state machine. There are multiple valid refinements of a task sequence. For example, there are different ways to navigate through a GUI in order to enter a new target in the navigation. The test case generator can be further parameterized by additional test selection criteria that define which paths should be chosen to refine a task sequence. The easiest solution is to generate a path that represents the shortest possible path that is a valid refinement of a given task sequence. There are many approaches [MAMS06, HKU04, HR04, AO99, RH01, BR04, LP04, EKRV06, Cho07, ZML07] for test case generation from state machines which we will not go into further detail here. Nevertheless, in our case study in Chapter 9, we introduce the approach that we applied for task sequence refinement.

6.4.1 Example for State-Machine-based Refinement

Figure 6.9 shows the mapping of a task model to a state machine that describes a part of an automotive GUI. A GUI is an example where the scenario-based refinement is not possible which we have shown in Section 6.3.3. Based on the state machine to task model mapping we are able to refine the task sequence Retrieve Traffic Info $\rightarrow$ Show Map Scenario with paths in the state machine. Figure 6.11(a) and 6.11(b) show two valid refinements of the task sequence. A path in the state machine is described by an MSC in which the signals between tester and system represent events, and the diamonds represent the corresponding state in the state machine. The two MSCs differ from each other because in the second MSC in Figure 6.11(b) the system returns into the main menu before the task Show Map is performed.

6.5 Summary

Task sequences lack two elementary properties of a test case: they describe no input and no output behavior. In this chapter we outlined two approaches of task sequence refinement by adding the missing behavior. The first approach uses scenario descriptions for refinement. The underlying assumption is that during testing and specification the typical tasks are described by scenarios. By mapping atomic tasks to scenarios we are able to refine task sequences. However, scenario-based refinement cannot be applied to all kinds of systems. When the input behavior of a task depends on the postcondition of the previously executed task, it is not possible to describe a task with one scenario. This problem is solved by the second approach we presented in this chapter: the state machine-based refinement. The idea behind our approach is to map a task model to a state machine that describes the system's behavior in more detail. Based on this mapping we are able to generate input sequence for a given task sequences that incorporates previously executed tasks.

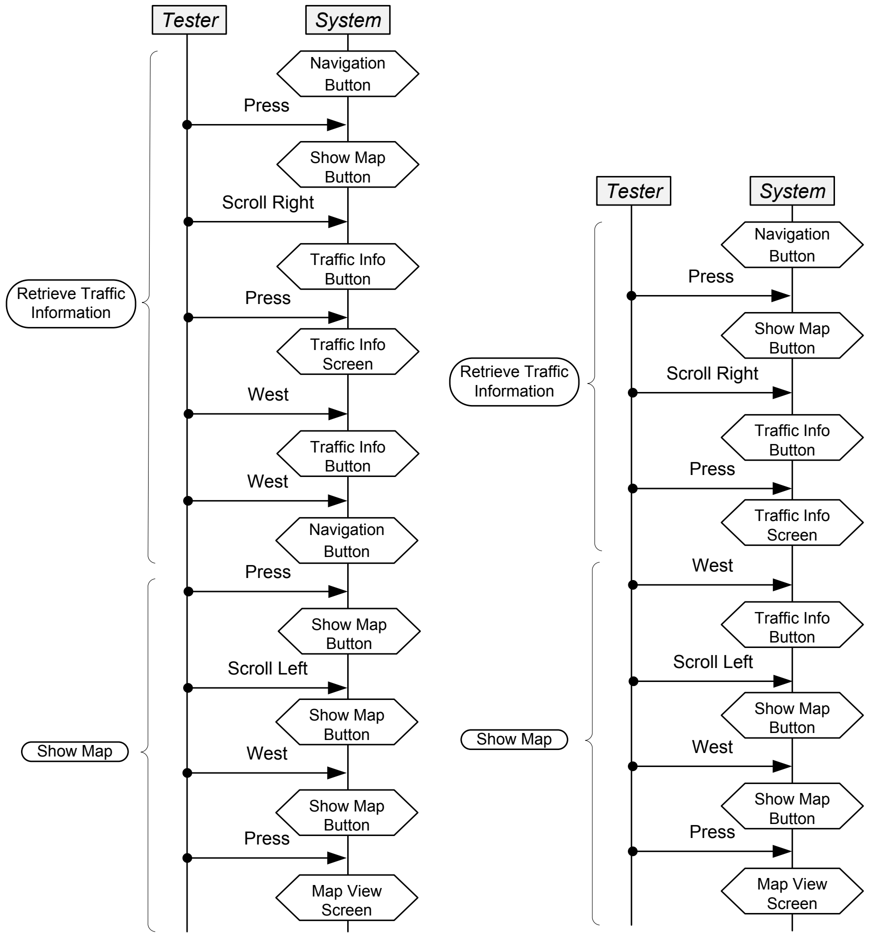


Fig. 6.11. State-based refinement of Retrieve Traffic Information → Show Map.

7

Generating Test Scripts using *AspectT*

In order to automatically execute the refined task sequences, they must be transformed into executable test scripts. We refer to the mapping of abstract test cases to executable test scripts as *test case instantiation* [PERH04], which is an elementary part of model-based testing. However, test case instantiation is not specific to our approach of task-based test case generation. Rather, all approaches of model-based testing are based on abstractions and therefore require the instantiation of the generated test cases. The approach presented in this chapter applies principles of aspect-orientation to the problem of test case instantiation in order to reduce the effort of test case instantiation. The contributions of this chapter are:

- A taxonomy of test case instantiation concerns.
- A modular approach for test case instantiation based on aspects.
- *AspectT*: A new aspect-oriented language for test case instantiation.
- A generic join point model for the eclipse modeling framework (EMF).
- An aspect weaver that enables test case instantiation for offline test script generation, test script generation during simulation, as well as on-the-fly test execution.
- The concept of aspect-oriented test case generation where aspects are used as test selection criteria.

This chapter is structured as follows. First, the different concerns of test case instantiation and the problems that they pose are discussed. Based on this, we present our approach of aspect-oriented test case instantiation. We then present our language, *AspectT*, and its weaver implementation. Subsequently, we briefly introduce an approach to aspect-based test case generation. The results section presents the impact of *AspectT* on the automated generation of test cases at BMW Group.

7.1 Testing Concerns

The instantiation of abstract test cases is an important part of the model-based testing process and is often time-consuming. Especially embedded systems require complex test setups to simulate the environment and to observe the behavior of the system. In the automotive infotainment domain, for example, the content of the graphical user interface is observed using screen grabbers, audio signals are observed by microphones, and haptic user inputs are simulated using special robots. Utting and Legeard performed case studies where test case instantiation took about 25-45% of the modeling time [UL06]. Test case instantiation is complicated by the fact that it varies depending on a number of concerns:

- One abstract test case is used to test different system properties: for example the same test case is used to test a GUI's timing behavior and its graphical representation.
- The current test phase (component, integration, acceptance test) affects the necessary driver components: for instance, component testing requires driver components that simulate the behavior of other components whereas integration testing requires driver components that simulate environment inputs.
- The testable system properties are restricted by the extent of implemented features at a certain stage of development.
- The input behavior of a test case has to be varied: for example, to discover race conditions the timing of test inputs must be varied.
- The test framework, driver components and testing environment must be configured and initialized depending on the test setup.

In this section we introduce different concerns of test case instantiation. We start with an example from the automotive domain where we used UML statecharts[1] to generate test cases. Figure 7.1 shows such a test model. The statechart describes the dialog behavior of an automotive GUI of a telephone application. The model is an abstraction of the real system where GUI and telephone are distributed across two different electronic control units and interact via bus communication. The example comes from the case study that we describe in Section 7.5.

Each path in the model is a potential test case. The bold transitions in Figure 7.1 indicate such a path for the Incoming Call use case. The initial state is the CD Player button (CD Player) in the main menu. When an incoming call occurs (Incoming_Call) the GUI shows the incoming call screen and the accept call button (Accept Call) is focused. When the user presses the IDrive controller (Enter) the call is accepted (Active Call). To finish the call, the user presses the IDrive controller again and the GUI returns to the CD player button (CD Player) which was saved by the history state (H). This path can be expressed by a hierarchical test case as shown in Figure 7.2. The test case

[1] http://www.omg.org/uml/

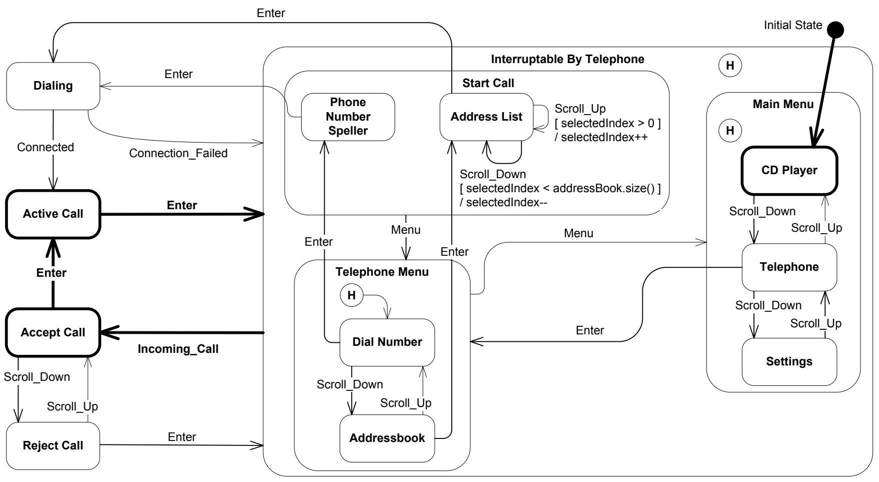


Fig. 7.1. GUI statechart model.

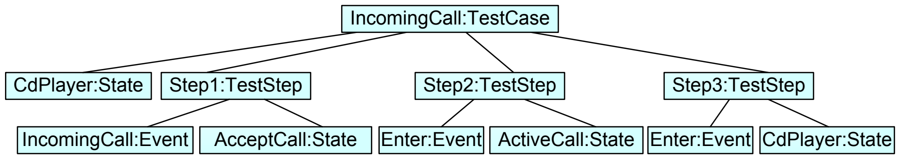


Fig. 7.2. Test case for the Incoming Call use case.

consists of an initial state and multiple test steps. Each test step has an input event and a postcondition in the form of a state. The actual execution of this test case requires additional information that bridges the abstraction gap between test model and the SUT. For a given input event the system must be stimulated and the abstract postconditions must somehow be monitored during test case execution.

Currently there are two main approaches to test case instantiation [UL06]: the adaptation approach and the transformation approach. The adaptation approach solves the problem of the different abstraction levels between test case and SUT by manually implementing a wrapper around the SUT: a driver component translates between the concrete level of the SUT and the abstract level of the test case. Transformation approaches translate abstract test cases into executable test cases and are typically based on model-to-text transformation frameworks. During the translation process the test cases are enriched with the missing information for bridging the abstraction gap. For instance, an abstract user input is translated to the actual bus message that represents this input. In practice, mixed approaches are common where driver compo-

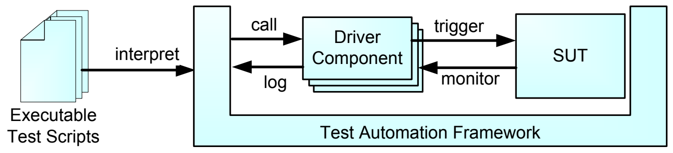


Fig. 7.3. Test automation framework and driver components.

nents raise the SUT's abstraction level and test cases are translated into test scripts that use these driver components during execution (see Figure 7.3). For testing infotainment systems at BMW Group, we use such a combined approach, where driver components encapsulate test triggers or test observers. We formulate the test scripts in a domain-specific test language based on Python.[2] The event Enter in the test case in Figure 7.2 is translated into the corresponding test script code that calls the IDrive controller driver component. The states in the test case are replaced with the corresponding oracle definitions that decide if the SUT is in the correct state. Listing 7.1 shows such a test oracle definition. The oracle tests if the currently focused button shows the correct text. *ScreengrabberService* is a driver component that provides services for analyzing the current screen content. The code snippet demonstrates a possible instantiation of the state CD Player.

```
buttonText = ScreengrabberService.getFocusedText()
if (buttonText != 'CD Player'):
  Log.error('Test failed: CD Player button was: '+buttonText)
```

Listing 7.1. Test oracle for CD Player state.

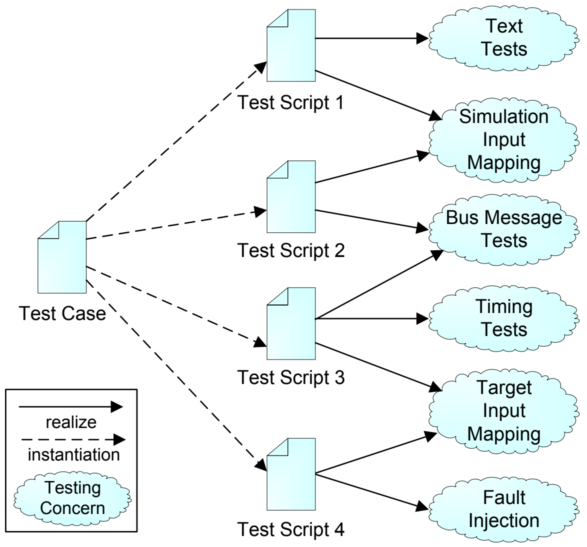


Fig. 7.4. One test case - many test scripts.

One test case can be used to test different system properties. A test generated from the example in Figure 7.1 can be used to test whether all buttons show the correct text, whether the correct bus messages have been sent and whether the GUI response is fast enough. All of these system properties represent the same model state. As a consequence, there are many possible mappings from a test case to a test script, and one chooses one of them based on the particular system properties that one is interested in testing. We refer to the information that describes the transformation of a test case to a test script as *test focus*. Figure 7.4 shows an example of dependencies between test case, test scripts and testing concerns. For a given test case, four test scripts are

[2] http://www.python.org

created that implement certain testing concerns. These concerns are different input mappings and test oracle definitions which depend on test focus and test setup. Figure 7.4 also shows that different test scripts share certain testing concerns, such as the bus message tests that are performed in a PC based simulation and on the target platform. Hence, testing concerns cut across different test focus definitions.

The question we answer in this chapter is how to define the *test focus* in such a way as to minimize the effort of test case instantiation and to enable an easy test focus adaptation to different testing contexts. Before we present the approach, we need to describe the factors that influence the definition of a test focus. During the development of a test case generation tool at BMW Group, we have identified several influencing factors for the *test focus* definition. We give a classification of these testing concerns in Figure 7.5. The main testing concerns are:

Refinement: The basic task in test case instantiation is to enrich test cases with additional information to bridge the abstraction gap to the SUT. The gap is determined by the model's abstraction level, which is chosen depending on the modeler's intention. The intention in our case study is to describe the menu structure and the navigation behavior of an automotive GUI. Therefore, the generated test cases can be used for testing the menu structure and the navigation behavior. The *refinement* is divided into *input mapping* where abstract test inputs are mapped to concrete test triggers, and *test oracle definition* where a certain abstract condition or state is mapped to an oracle that observes the SUT. In addition, test cases can also be used for testing additional properties that are beyond the original intention of the model, such as the communication between a GUI and other control units. The latter is often the case for non-functional requirements such as timing constraints. These *additional constraints* are often not included in the model because they

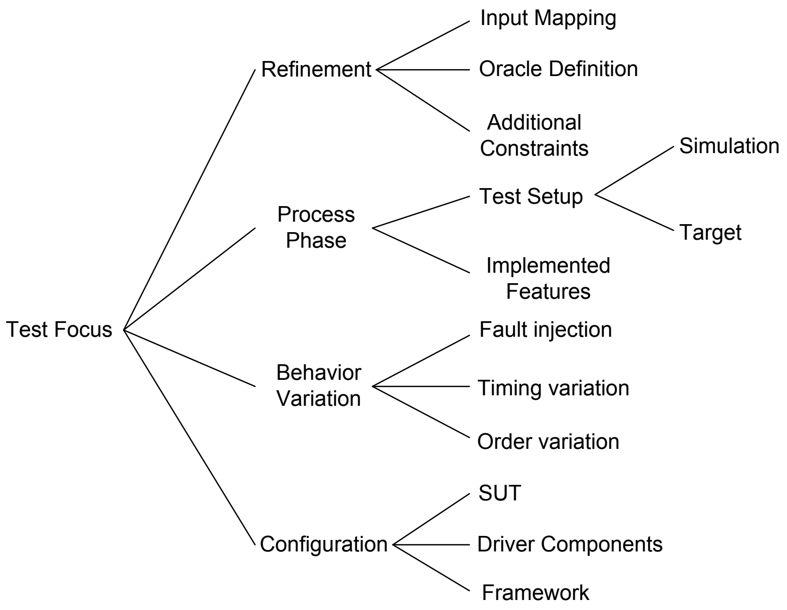


Fig. 7.5. Classification of test instantiation concerns.

cannot be expressed in the used modeling language or because the model has another abstraction level.

Process phase: Typically, testing is divided into component testing, integration testing and acceptance testing. In each test phase, specific input mappings and oracle definitions are necessary depending on the *test setup*. For example, different driver solutions are necessary depending on whether the SUT's environment is simulated or real. With increasing development progress the system properties that must be tested become more complex. Hoever, the test model often stays at the same abstraction level. As a consequence, the test oracle definition becomes more complex. Another important factor is the extent of *implemented features*: certain features are implemented earlier and can therefore be tested earlier.

Behavior variation: Especially if the system interacts with third party software or human users, it is important to test how the program behaves on non-specified inputs. This can be achieved by the injection of *faults* and undefined inputs or by the variation of an input sequence. In concurrent systems, for instance, it is important to vary *timing* or *order* of message sequences to test for race conditions.

Configuration: Test cases must often be enriched with additional code for initialization and configuration of the *test framework*, the *driver components* and the *SUT*.

These factors just described, represent testing concerns that crosscut multiple *test focus* definitions. To take advantage of the potential of model-based testing, the chosen approach for test case instantiation must be able to handle the introduced testing concerns. To reduce the effort of test case instantiation, it is necessary to encapsulate a testing concern in order to reuse it in different *test focus* definitions.

7.2 Aspect-based Test Focus Definition

In this section we present our approach of aspect-oriented test case instantiation. Our starting point is an abstract test case that has been derived from a behavior model (see Figures 7.1 and 7.2) or manually created in a dedicated modeling language. This test case is transformed during test case instantiation into a test script. This transformation is a typical application of model-based code generation. In the previous section we introduced the concept of a *test focus* that defines such a transformation of a test case into a test script. As a consequence, each test focus requires a separate code generator. In the last section, we introduce several testing concerns that crosscut multiple test focus definitions. This leads to the problem of finding the right modularization of these crosscutting concerns to enable their reuse in different test focus definitions.

What is needed is a way of describing the test oracle definitions, test trigger definitions and configuration information that allows us to apply this information to any potential test case for a given test model. Each of these testing

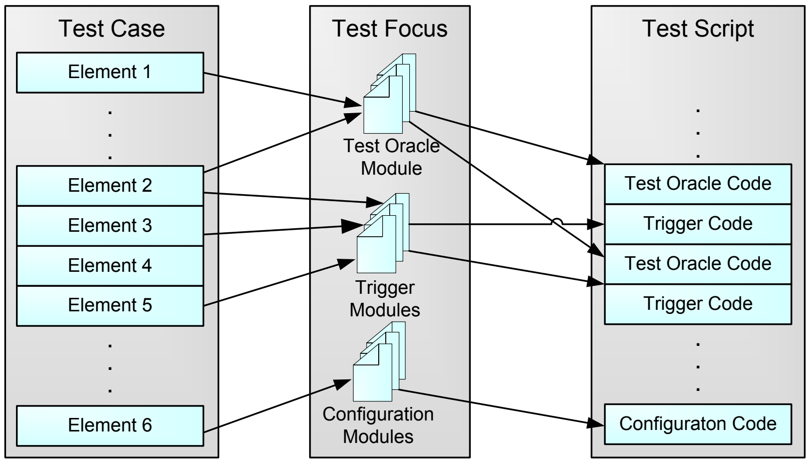


Fig. 7.6. Transformation of a test case into a test script.

concerns should be encapsulated in a separate module. During test case instantiation, one element in a test case can be instantiated by multiple modules. For example, a state may require two different test oracles, where each oracle checks a specific system property that represents this state. On the other hand, one test oracle can apply to different states in a test case. For instance, a text test oracle can be applied to any state that represents a button. Figure 7.6 shows these relations between test focus modules, test case and test script.

Aspect-oriented software development approaches [Kic96] solve this problem by encapsulating crosscutting concerns in aspects. Such an aspect is composed of pointcuts and advice. A pointcut defines locations in a base program or base model where an advice is going to be inserted.

In our approach we regard each testing concern as an aspect that can be woven into different test cases. For instance, the crosscutting concern Incoming Call Input Mapping is defined in an aspect that encapsulates the corresponding telephone driver component calls. These can be woven into the different locations in a test case where an incoming call input event occurs, as depicted in Figure 7.7.

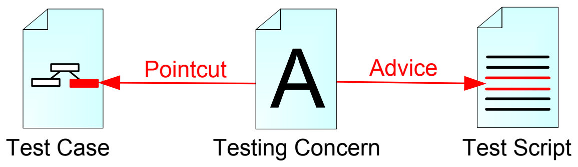


Fig. 7.7. Aspect-oriented testing concern.

In common aspect-oriented approaches the base is an existing program or model into which aspects are woven. The final result is the composition of base and aspects. In test case instantiation, the test case acts as the base. We produce a test script from such a test case by a tree walk through the test case's abstract syntax tree (Figure 7.2). Each node in the test case tree is a potential join point where aspects emit code fragments into the test script. The resulting test script is the composition of all code fragments that were emitted during weaving.

Similarly to common aspect-oriented programming (AOP) approaches, such as AspectJ, our aspect definition consists of pointcuts and advice. An advice contains the test script code that is emitted into the test script. Each advice is assigned to a pointcut that selects the join points in a test case where the advice should be inserted. Some aspects define the base implementations for test case entities. The base implementation of the event IncomingCall in Figure 7.2 is defined by an advice that implements the incoming call trigger. We call this a *base advice*. Other aspects represent additional constraints, such as timing constraints. The advice that implements additional constraints or triggers is invoked either *before*, *after* or *around* a join point.

In the previous section, Listing 7.1 showed an instantiation of the test oracle for the state CD Player Button. In our aspect-oriented approach, the test script code from this listing is part of an advice definition. This advice is assigned to a pointcut definition that selects all CD Player Button states in a test case. Each state is a node in the test case tree. The kinds of nodes that can occur in a test case tree, such as State or Event in Figure 7.2, are defined by the test case's metamodel. Thus, a test case's metamodel represents the join point model and each entity in the metamodel is a join point. Test case metamodels differ depending on the modeling language, the test case metamodel for test cases defined as MSCs is different from the metamodel of test cases derived from a statechart. Therefore, a pointcut can define multiple join point selections in order to support multiple test case metamodels. Thus, one test oracle can be reused to instantiate test cases in different modeling notations.

Additional constraints, such as timing constraints, must often be implemented by multiple advice. Timing constraints are defined by two advice: one that starts a timer before an event is triggered and one that stops it when the desired state is reached. One example of a timing constraint is the following: "The state Warning Message Screen should be reached within 100ms of an occurrence of the event Low Fuel." However, if a generated test case reaches the state Warning Message Screen by the input event Low tire pressure instead of Low Fuel, the test script execution would result in an exception. The advice Start Timer has not been weaved and therefore the timer initialization code would have not been emitted to the test script. This would result in an exception when Stop Timer is executed, because the timer would not have been initialized. As a consequence, our language for the *test focus* definition provides means to describe such inter advice dependencies: advice can define *preconditions* and *postconditions* in the form of other advice that must have been woven before or afterward. For the timing example mentioned above, the advice definition for Stop Timer is extended by the statement *precondition* Start Timer, which adds the constraint that Stop Timer can only be woven if the advice Start Timer has been woven before.

By modularizing testing concerns into separate aspects the *test focus* definition is reduced to defining a set of aspects. Each trigger definition, oracle definition or configuration definition must be specified once and can then be reused in any other test focus definition.

7.3 The Test Case Instantiation Language *AspectT*

The previous section described the general concept of aspect-oriented test case instantiation. In this section, we introduce our implementation of the concept: *AspectT*. In the first part of this section the language and its concepts are introduced. The second part presents the implementation of the weaver.

7.3.1 The Join Point Model

Different testing concerns are encapsulated in separate aspects. The aspects are woven into an abstract test case. Therefore, each entity in a test case is a potential join point. The different entities and their relations are defined in a test case's metamodel. There is no general test case metamodel because it depends on the underlying modeling language. For example test cases for communication behavior are specified using MSCs. In model-based test case generation, test cases are created from UML based notations, constraint logic programming or model checking. Currently domain-specific language (DSL) are gaining influence in development, specification and hence in model-based test case generation. The case study presented in this chapter was originally defined using a DSL for the specification of automotive user interfaces. All these approaches have in common that the resulting test cases primarily represent the characteristics of the underlying modeling language.

In order to avoid the definition of a general-purpose test case language and the implementation of a transformation that translates existing test cases into the standard test case model, we chose a generic approach for the join point model of *AspectT*. The approach in *AspectT* is to define the join point model at a modeling language independent level. For UML based notations, this would be the meta object facility (MOF) [Obj02] which is used for defining the different UML diagrams. MOF describes only the abstract syntax of these modeling diagrams. In order to be more flexible, we chose the Ecore [BBM03] language which is part of the EMF. Ecore implements a subset of MOF and is tightly integrated into the Eclipse IDE.[3] The advantage of EMF is the mature tool support. There are, for example, several frameworks to define graphical editors or textual editors based on an Ecore model. The standard UML notations are supported by using the EMF based UML2 implementation.[4] An Ecore model consists of entities (instances of *EClass*) and their relations expressed by composition, inheritance and association. Using these constructs it is possible to define the abstract syntax of a test case such as the one shown in Figure 7.2. Figure 7.8 shows an example test case metamodel for state machines as it is defined in Ecore. Each entity (TestCase, State,...) is an instance of EClass.

A test case is executed sequentially. As a consequence, the test case metamodel must describe an ordered structure. To describe complex test case entities (e.g., a test step that is composed of an event and a state), we chose a tree structure in *AspectT*. The test case sequence is defined by a depth-first left-right tree walk. A test case is either described by a dedicated test case

[3] http;//www.eclipse.org

[4] http://www.eclipse.org/modeling/mdt/?project=uml2

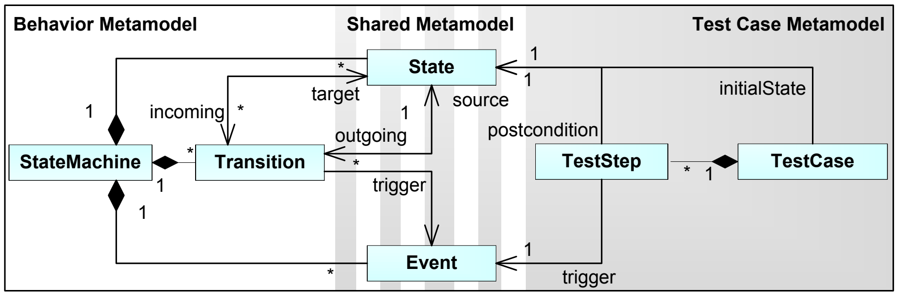


Fig. 7.8. Test case metamodel.

metamodel, or on top of a behavior model's metamodel, where the leaves of the test case tree are the entities of the behavior model. An example of the latter is shown in Figure 7.8, where the root of the test case metamodel is a test case. A test case consists of multiple test steps, where each test step has a trigger and a postcondition. The trigger is an *Event* and the postcondition is a *State*, where both are defined in the statechart metamodel. This is an important advantage of domain-specific modeling, because we are able to combine a structural model, a behavioral model and a test case model depending on our domain. Using these dependencies one can retrieve additional information for a state from a corresponding structural model to define more detailed test oracles. We give an example of such a domain model in Section 7.5. Figure 7.9 shows the relationship between Ecore, behavior model and test case model.

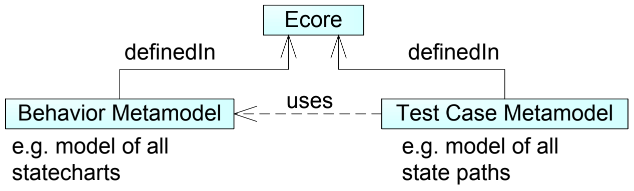


Fig. 7.9. Behavior and test case model are defined in Ecore.

By using Ecore models as join point models, it is possible to use *AspectT* with different test case models. Hence, the same aspect can be used for the instantiation of test cases that are defined in different modeling languages. An aspect can define multiple pointcuts that select join points from different test case models, as depicted in Figure 7.10. Furthermore, existing test case notations can be integrated by defining a corresponding metamodel in Ecore and a parser that instantiates the model. We performed this to instantiate test cases that were generated using the model checker SPIN [Hol97]. The downside of using a metamodeling language like Ecore is that it describes only the abstract syntax. Thus, the semantics of the language are not formally defined. Therefore, it is only possible to define pointcuts based on the abstract syntax. A test case model with specific semantics would enable the definition of pointcuts based on specific test case criteria. But we deliberately chose to base *AspectT* only on the abstract syntax in order to be more flexible with respect to integrating new test case models. In the following subsections we introduce the different parts of *AspectT* in more detail.

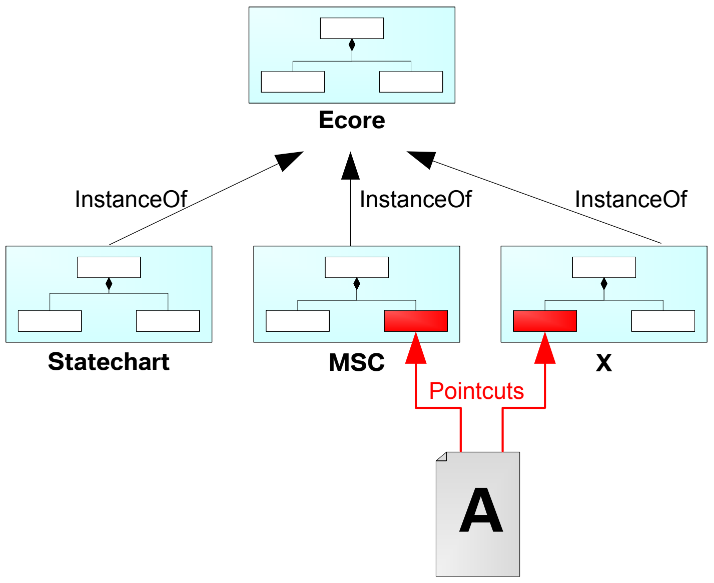


Fig. 7.10. Defining aspects for different modeling languages.

7.3.2 The Aspect Definition

The *test focus* describes a testing concern specific transformation of a test case into a test script. In *AspectT* a testing concern is encapsulated in the form of an aspect. Thus, we can define a *test focus* by a set of aspects. Each aspect has one or more advice which implement a specific testing concern. An advice describes a specific part of a test script. There are different languages for test script definition, such as general-purpose languages (e.g., Java, Python,...) or DSLs (e.g., TTCN-3 [GHR$^+$03b]). For testing infotainment systems at BMW Group we use a DSL that is an extension of Python. To support any test script language, an advice is defined in *AspectT* by a metaprogramming language that enables the generation of source code in any test script language. The join point at which an advice is woven into a test case is selected by a pointcut. A pointcut selects a join point by defining an instance of *EClass* and by additional constraints on the actual join point object.

To give an overall impression on *AspectT*, Listing 7.2 shows an example aspect definition. The weaving happens during a tree walk through the test case. The aspect writes for the first time it is woven:

```
print 'Hello World!'
```

Any additional time it is woven, it writes the state's name and the text "has already been woven". The pointcut SampleState selects any instance of *State* that has the name sample. The latter is verified by a constraint statement that checks if the state's attribute name is equal to sample. The advice contains the program that controls the text that is written to the resulting test script.

```
aspect HelloWorldAspect{
  def hasBeenWeaved = <% false %>
  pointcut SampleState{
    State : self.name = 'sample'
  }
  advice HelloWorldAdvice after SampleState<<
  <% if (!hasBeenWeaved) {
      hasBeenWeaved = true %>
    print 'Hello World!'
  <% } else {%>
    print '<%=self.name %> has already been woven'
  <% } %>
  >>
}
```

Listing 7.2. Sample aspect definition.

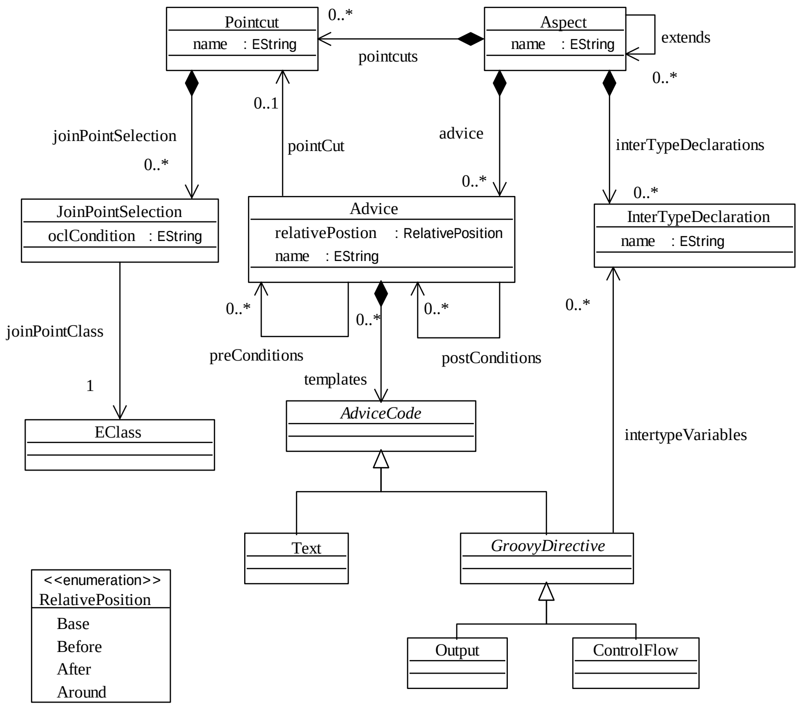


Fig. 7.11. The Ecore metamodel of *AspectT*.

The *AspectT* tool has also been implemented using EMF. Figure 7.11 shows the metamodel of *AspectT* as defined in Ecore and its relation to Ecore: the *joinPointClass* reference between *EClass* and *JoinPointSelection*. In the following sections the elements of *AspectT* are introduced in more detail.

Pointcuts:

Pointcuts select join points from a test case. A join point selection is defined by the join point's class (e.g., State) and by an additional constraint on the join point object. Figure 7.12 shows an example for a join point selection. A pointcut can contain multiple join point selection statements. The constraint

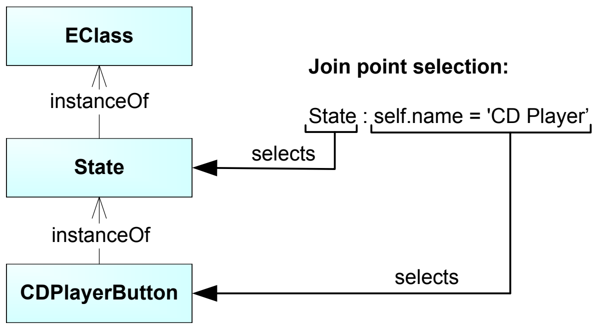


Fig. 7.12. pointcut definition in *AspectT*.

is defined using the object constraint language (OCL),[5] where *self* refers to the current join point object. OCL was chosen due to its powerful query expressions.

Advice:

Advice are woven into a test case and emit join point specific text into the resulting test script. The following join point adaptation kinds are supported:

- **before:** The advice emits its text before a join point.
- **after:** The advice emits its text after a join point.
- **base:** The advice is the base implementation of a test case entity.
- **around:** The advice emits its text before and after a join point. The *proceed* tag separates the advice into the parts that are written before the join point and the ones that are written after the join point. *Around* advice are internally divided into a *before* and an *after* advice. Only if the adaptation kind is *around*, the *proceed* tag is allowed in an advice definition.

Advice are defined by a metaprogramming language, which is based on the scripting language Groovy[6]. Using this language it is possible to generate source code depending on the current join point. The concept is similar to embedded script languages such as JSP[7] or PHP.[8] An advice defines text that is emitted into a test script. The language features the following concepts:

- The text between "<<" and ">>" is emitted into the test script.
- The text inside an advice can additionally be controlled using directives, where "<%" and "%>" indicate the start and end of a directive. Directives are Groovy blocks that control the text which lies between. In the example in Listing 7.2 we used directives to write the text depending on the

[5] http://www.omg.org/docs/ptc/03-10-14.pdf
[6] http://groovy.codehaus.org/
[7] http://java.sun.com/products/jsp/
[8] http://www.php.net/

variable hasBeenWeaved into the resulting script. Directives may contain any Groovy code. Inside an directive it is possible to access the join point object using the *self* variable. Via *self* it is possible to retrieve any data from the test case. For example, if *self* references a state, *self.outgoing* selects all outgoing transitions. The outgoing reference is defined in the test metamodel which is shown in Figure 7.8.

- Dynamic data can be written to the resulting script using the "<%=" and "%>" tags. The result of the expression inside these tags is written into the test script.

Advice Dependencies: There are test oracles that must be implemented in multiple advice. This applies for constraints that affect multiple join points. If an advice depends on another advice (e.g. by an intertype variable), the advice can only be executed if the other advice has been executed before. This relation must be defined at advice level rather than at pointcut level because they depend only on the advice implementation; therefore, they are independent of the pointcuts. The explicit definition of these dependencies is necessary, because it is not guaranteed whether a test case contains join points for the pointcuts of both advice. Advice that must have been woven before are defined as *preconditions*. *Postconditions* indicate that the specified advice must be woven eventually after the current advice.

Aspects:

An aspect encapsulates a testing concern by defining different pointcuts and advice. Each pointcut may have one or more assigned advice. An aspect is always instantiated as a singleton.

Intertype Declarations: Inside an aspect, variables can be defined that can be accessed from within each advice. Intertype declarations enable the data exchange between different advice. The concept of intertype declarations is similar to that in AspectJ.

Aspect Inheritance: An aspect can inherit advice and pointcuts from other aspects. Superaspects are defined using the keyword *extends*. In this case all superaspects' advice are woven as well. Additionally an aspect can define its own advice for pointcuts defined in the superaspect. It is possible to define precedence relations between advice in the subaspect and in the superaspect. Aspect inheritance allows the reuse of pointcuts and advice across different aspects. Figure 7.13 shows an example of two aspects and their dependencies. BusService is the superaspect of IncomingCall. Advice in IncomingCall are now able to adapt pointcuts from BusService and to define advice dependencies to advice in BusService.

7.3.3 The *AspectT* Weaver

The aspect weaver input is a test case model and a test focus. The test case model must conform to its metamodel defined in Ecore. The metamodel must describe a tree structure (see example in Figure 7.2). The test focus is a set of aspects. There are three different weaving scenarios:

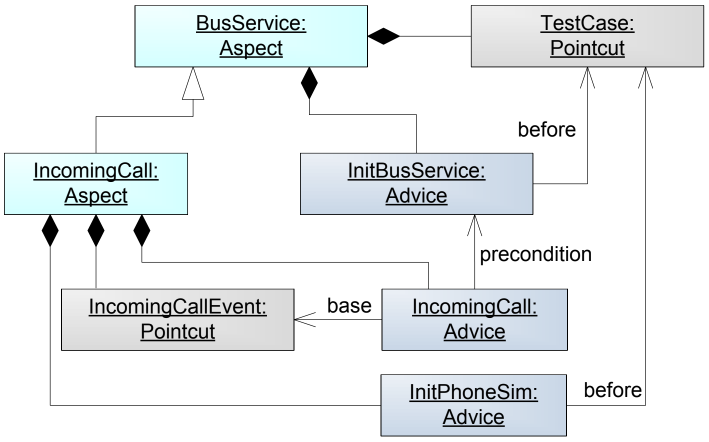


Fig. 7.13. Aspect and advice dependencies.

Offline Weaving: The weaving takes place after the test case has been generated. The weaver iterates over the test case and its child entities corresponding to a depth-first left-right walk and evaluates the pointcuts at each entity and evaluates a given advice if applicable. The advice are woven depending on their adaptation kind: *before*, *base*, *after* or *around* the join point. If two advice A and B are woven before the same join point P, the weaving order is not defined. If A is a *precondition* of B or B is a *postcondition* of A, then A is woven before B.

Online Weaving: In certain scenarios the offline weaving cannot be applied. For example, when the test case generation is performed by the execution of a state machine. Certain information is only present during statechart execution time, such as variable values in a certain state. Often these values must be inserted into the advice code to verify a specific system property. If the test case does not contain the actual values of all variables for each state, the variable values cannot be used for the definition of a test oracle. The solution is to weave the advice during the simulation of the behavior model. Thus, the test case is created during the simulation of the statechart. The weaver is invoked each time a test case element is created. The creation order of the sample from Figure 7.2 is:

IncomingCall(TestCase) → CdPlayer(State) → TestStep1(TestStep) → IncomingCall(Event) → AcceptCall(State) →...

Before and *base* advice are woven directly for each join point. The advice which must be inserted after a join point are saved in a stack. If the next join point is not a child of the previous one, the stack is emptied and the advice subsequently emits its text into the test script.

Runtime verification: Another use case is runtime verification, where test case creation and test script execution happen at the same time. For example, a test generator generates random input events for the statechart in Figure 7.1. For each event a new test step is created that contains the event and its post condition. A test focus, consisting of multiple aspects, is weaved into the test step (similar to Online Weaving). The resulting test script code for the test step is directly executed which results in the stimulation and monitoring of the SUT. In this special case the *postcondition* advice dependency is ignored, because it is not possible to determine if a required advice (indicated by *postcondition*) will be executed.

7.4 Using *AspectT* for Test Case Generation

In the previous sections, aspects have been used to translate test cases into test scripts. In this section we show that aspects can also be used as test selection criteria in model-based test case generation. A *test focus* highly depends on the test generation with its test selection criteria. Especially in large systems with different test suites, which require different test setups and test drivers, a *test focus* represents a set of specific system properties. In an automotive GUI, one useful *test focus* is to test whether all buttons' texts are shown correctly. The corresponding test oracle TextTest is shown in Listing 7.3. The pointcut selects all *states* that represent a button. A button state must hold: *self.widget.type = 'button'*.

```
aspect TextTest{
  pointcut Button{
    State : self.widget.type = 'button'
  }
  advice ListTextTest base Button<<
    ScreenGrabberService.testButton('<%=self.widget.text %>') >>
}
```

Listing 7.3. Text test aspect.

The test trigger definitions are shown in Listing 7.4. The system is stimulated using the IDrive controller, where for each input a pointcut is defined. Using these aspects we can translate a generated test case into a test script for text testing.

To reduce the effort of test execution, generated test cases should cover only the relevant system properties. In the example presented above, the generated test sequence should be a minimal menu walk that covers all text buttons and as few non-text-buttons as possible. The test selection criterion for this test case is therefore that only states that represent a button should be selected. This is identical to the pointcut definition of the aspect TestText in Listing 7.3 which selects all buttons. Therefore, each pointcut is a potential test selection criterion. An example of another use case is a test setup that supports only a limited set of possible triggers. A simple test setup for the case study would only support test inputs in the form of IDrive controller inputs and no inputs from other applications, such as the telephone. In this case we have to add another test selection criterion that restricts the possible transitions to the ones that can be triggered by the IDrive controller. This test selection criterion corresponds to the pointcuts in the trigger aspect in Listing 7.4.

```
aspect IDrive{
  pointcut Enter{
    Event : self.name = 'Enter'
  }
  pointcut Menu{
    Event : self.name = 'Menu'
  }
  pointcut Left{
    Event : self.name = 'Scroll_Up'
  }
  pointcut Right{
    Event : self.name = 'Scroll_Down'
  }
  advice IDriverEnter base Enter <<
    IDriveService.Enter() >>
  ...
}
```

Listing 7.4. IDrive controller aspect.

When we combine the pointcut definitions of the IDrive aspect and of the TextTest aspect, we have the ideal test selection criteria for the test case generation from a statechart. The test generation algorithm must be able to generate a minimal path that contains all states that are selected by TextTest where only transitions are used that can be triggered by the events defined by the pointcut of IDrive. It depends on the behavior model if such an algorithm is possible and there is no general applicable solution. Nevertheless, this shows another potential benefit of aspect-based *test focus* definition.

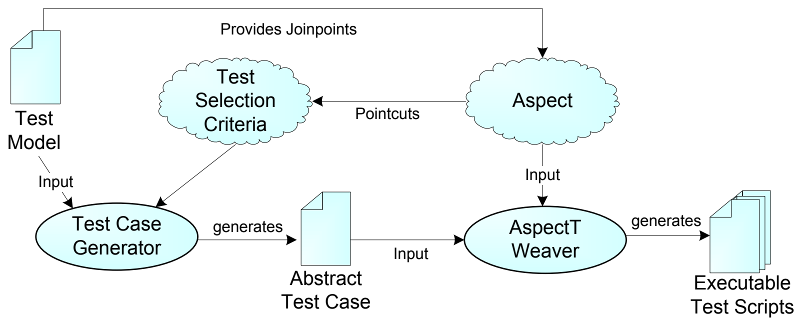


Fig. 7.14. Aspect-oriented test case generation.

Figure 7.14 shows the resulting process of aspect-oriented test case generation. The aspects are defined based on the join points that are defined by the test model. There are general test selection criteria, such as state coverage or transition coverage for the statechart based test case generation. These can be combined with *AspectT*'s pointcuts to generate test focus specific test cases. The resulting abstract test cases are finally translated into executable test scripts using the *AspectT* Weaver.

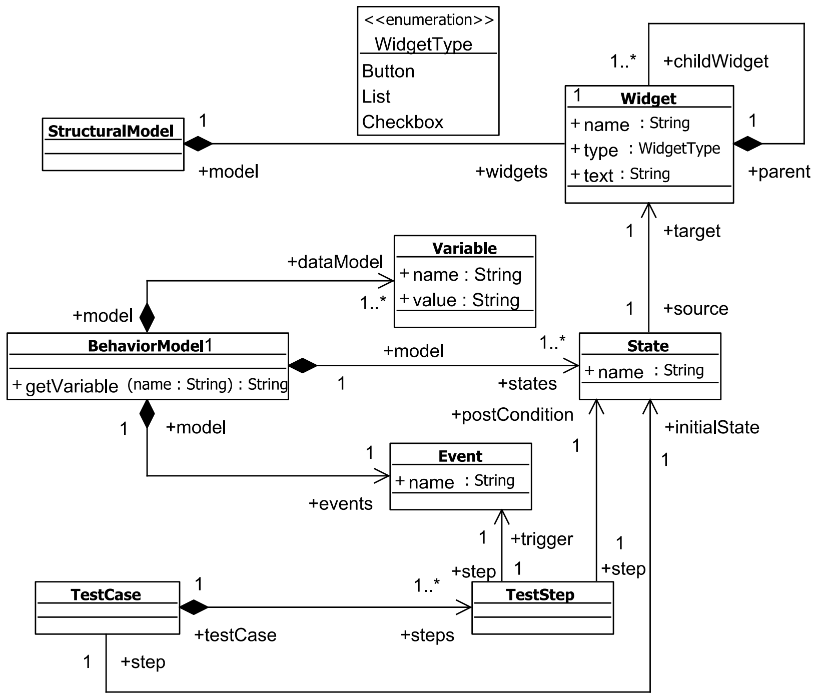


Fig. 7.15. Part of the case study's metamodel.

7.5 Example

In this section we introduce a small case study for the application of *AspectT* in the automotive infotainment domain.

7.5.1 Statechart based Test Case Instantiation

The case study is part of a GUI of an automotive infotainment system that controls a telephone application. The electronic control unit (ECU) that contains the GUI application is connected with the telephone ECU by a communication bus. The GUI is specified by a structural model that describes the different screens and their widgets. Widgets have properties: name, type (e.g. button, check box, list,...) and text. The dialog behavior of the GUI is defined using UML2 statecharts. The guards and actions in the statechart are defined based on a data model that consists of different variables (e.g. an address list). We introduced the behavior model earlier in Figure 7.1. The behavior model is linked to the structural model: each state refers to its corresponding widget. For example the state CD Player is linked to a widget CD Player Button that is of the type Button and contains the text CD Player. A test generated from the behavior model is a path in the statechart. Such a test case has an initial state and an ordered list of steps. Each step has a trigger and a postcondition, where the trigger refers to an event. Initialstate and postcondition refer to states in the behavior model. The structural model, behavior model and test case model are described in one Ecore model. By linking the different models it is possible to retrieve detailed information for a given state in the behavior model. For example we can get the widget for a given state and get additional structural properties that are important for the test oracle definition.

Based on the dialog model, a simple test case is generated for the incoming call use case. The resulting path in the statechart is marked bold in Figure 7.1. First we define a basic set of aspects to trigger the SUT by IDrive controller inputs and to monitor the SUT by checking the shown tests. Most of the test aspects require access to the bus system. Therefore we start with the bus configuration aspect: BusService in Listing 7.5. In this aspect it is clearly visible why we model a test case as a tree. The initialization and shutdown of a service can be woven at the start and at the end of a test case by defining an around advice for the pointcut Test Case.

```
aspect BusService {
  pointcut TestCase{
    TestCase
  }
  advice InitBusService around TestCase
  <<BusService.init()
  <%proceed%>
  BusService.shutdown() >>
}
```

Listing 7.5. The bus service configuration.

In the case study the events are triggered by a driver component that sends the corresponding bus messages. Listing 7.6 shows the aspect definition for the input mapping. The IDrive controller service relies on the bus service. Therefore the IDriveEnter advice has the inter advice dependency *precondition* to the advice InitBusService.

```
aspect IDrive extends BusService{
  pointcut Enter{
    Event : self.name = 'Enter'
  }
  advice IDriveEnter base Enter precondition InitBusService
   <<IDriveService.Enter() >>
  //...
}
```

Listing 7.6. Input Mapping.

In order to decide if the system is in the correct state, the text of the currently focused button in the SUT is compared with the text from the model. The text is checked by a screen grabber in combination with optical character recognition (OCR) software. The aspect that implements the corresponding test oracle is shown in Listing 7.7. The pointcut selects all instances of *State*. The additional OCL statement in line 4 selects only states of the type button. The pointcut uses the references between behavior model and structural model to retrieve the type and text information of the corresponding widget. The text is passed in the advice to the screen grabber component that checks if the focused element shows the correct text. To reduce the test case execution time each button is only tested once. This is accomplished by using an intertype declaration: *testedTexts*. The variable *testedTexts* is initialized below the aspect definition. The Groovy block in the ListTextTest advice uses this variable to check if the actual join point has already been tested. If a button has not been tested before, the test oracle code is written and the button's name is added to the variable *testedTexts*.

```
aspect TextTest{
  def testedTexts = <% new LinkedList<String>() %>
  pointcut Button{
    State : self.widget.type = 'button'
  }
  advice ListTextTest base Button<<
    <% if (!testedTexts.contains(self.name)) {
      testedTexts.add(self.name) %>
ScreenGrabberService.testButton('<%=self.widget.text %>')
  <% } %>
  >>
```

Listing 7.7. Test oracle for text tests.

The next example is an aspect that requires the online weaving approach. We want to test if the address book shows all addresses correctly. The address book is a dynamic list. Users can scroll through the list using the IDrive controller. The address list is contained in a variable addressBook, and another variable selectedIndex tracks the index of the currently focused address. These variables are defined in the underlying behavior model. The outgoing transitions of the state Address List increase or decrease the value of selectedIndex when the events Scroll_Up or Scroll_Down occur. As a consequence, the value of selectedIndex changes during the execution of a test case. When the state Address List is active, the selected index varies depending on the history. But the test case model does not contain the values of selectedIndex for each test step. Therefore, this aspect cannot be woven into an existing test case. Instead we simulate the statechart and create the test case dynamically. Each new test step is woven when it is created thus each test step can access the correct variable value.

```
aspect AddressBookTextTest{
  pointcut AddressState{
    State : self.widget.type = 'Address List'
  }
  advice TestListBehavior base AddressState<<
ScreenGrabberService.testButton(
'<%=self.model.getVariable('addressBook').getAt(self.model.getVariable
     ('selectedIndex')).value %>')
  >>

}
```

Listing 7.8. Testing dynamic list behavior.

The aspect in Listing 7.9 demonstrates how we can enrich an existing test case with additional constraints. When an incoming call occurs the GUI should show the incoming call screen within less than 100ms. The aspect TimingTest describes the test oracle for this timing constraint. We extend the aspect IncomingCall by the aspect TimingTest in order to reuse its pointcut definition. After the incoming call event, the StartTimer advice emits code that starts a timer. The advice StopTimer is invoked before the state AcceptCall. The test oracle defined by this advice waits for the next screen to show and checks if it occurred within 100ms using the timer created in StartTimer. The advice StartTimer and StopTimer depend on each other, because in the first advice a timer is created, which is stopped in the second advice. The dependencies are defined by the additional *postcondition StopTimer* and *precondition StartTimer* statements. However, defining advice precedences only guarantees that an advice has been woven before or after another advice. There are no guarantees during test script execution if the advice is actually executed before. The basic assumption is that test scripts are executed sequentially. The resulting test script triggers the incoming call (IncomingCallTrigger), starts the timer (StartTimer) and finally waits for the correct state of the GUI (StopTimer).

```
aspect IncomingCall{
  pointcut IncomingCallEvent{
    Event : self.name = 'Incoming_Call'
  }
  advice IncomingCallTrigger base IncomingCallEvent
  <<TelephoneService.triggerIncomingCall() >>
}
aspect TimingTest extends IncomingCall{
  pointcut AcceptCallState {
    State : self.name = 'Accept Call'
  }
  advice StartTimer after IncomingCallTrigger postcondition StopTimer
  <<Timer.start('IncomingCall')>>

  advice StopTimer before AcceptCallState precondition StartTimer<<
ScreenGrabberService.waitForText('<%=self.widget.text%>')
if (Timer.stop('IncomingCall') > 100):
  Error.log('Timing failed')
  >>
}
```

Listing 7.9. Timing test.

The next example demonstrates the behavior variation using *AspectT*. The pointcut condition is fulfilled for each state that has no outgoing transition for the scroll down event Scroll_Down. The test script in the advice tests if the text of the focused element changes, when the event Scroll_Down is triggered and logs an error with the state's id. An illegal state change is detected when the focused text changes.

```
aspect BehaviorVariation{
  pointcut State{
    State : not self.outgoing.event->exists(i|i.name='Scroll_Down')
  }
  advice TurnLeft before State <<
text = ScreenGrabberService.getText()
IDriveService.triggerLeft()
if (text != ScreenGrabberService.getText()):
  Error.log('<%=self.name%> Text changed on IDrive left')
  >>
}
```

Listing 7.10. Example for behavior variation.

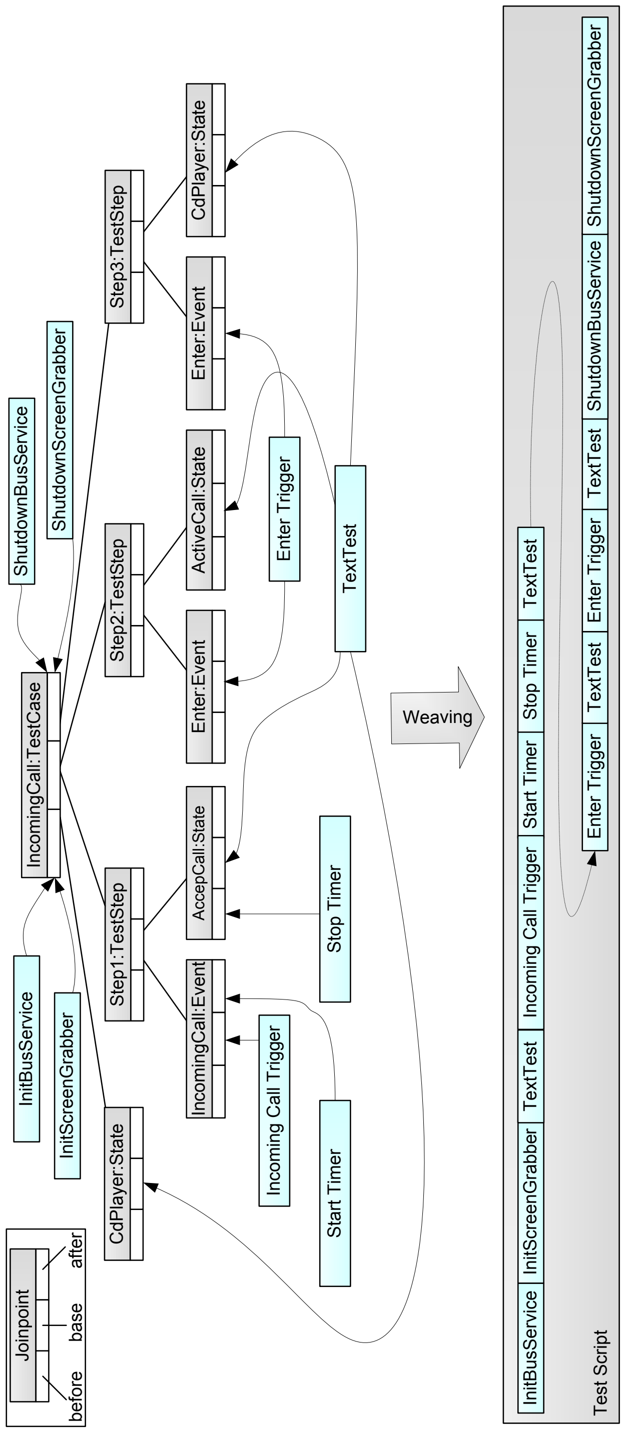


Fig. 7.16. AspectT weaving.

Each aspect presented above implements a separate testing concern, either input mapping, test oracle definition or configuration code. We also showed how *AspectT* can be used to define additional constraints and how to inject faulty inputs. By separating these concerns the test focus definition is reduced to selecting a set of aspects. Using these aspect definitions we are able to define different test focuses:

- **Text:** The goal is to test whether all buttons show the correct text. Therefore we need the IDriver aspect and the aspect IncomingCall to trigger the system in order to reach all states. The test oracle is defined in the aspect TextTest.
- **Timing:** The goal is to test if the system fulfills certain timing constraints, for example, the one defined in the aspect TimingTest. The test focus additionally requires the IDrive aspect to trigger the system.
- **Text and Timing:** It is also possible to combine the previous test focuses to test text and timing in one test script. In this case the Text test focus is extended with the TimingTest aspect.
- **Undefined inputs:** This test focus exercises the SUT with undefined inputs. Undefined inputs are any IDriver controller inputs that trigger no transition in a certain state. The test focus is defined by the aspects BehaviorVariation,IncomingCall and IDrive.
- **Address book test:** This test focus tests the dynamic address book behavior which requires the aspects IDrive and AddressBookTextTest.

Figure 7.16 shows the weaving of the *Text and Timing* test focus into an abstract test case and the resulting test script.

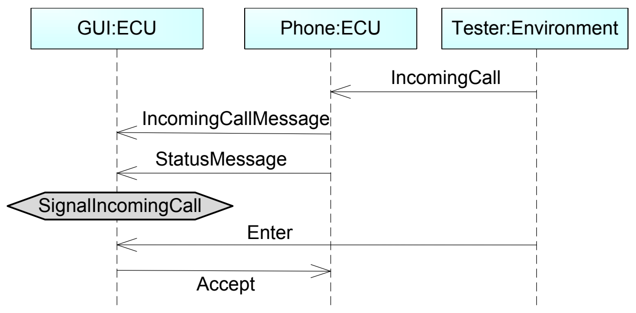


Fig. 7.17. MSC based test case.

7.5.2 MSC based Test Case Instantiation

In the previous paragraphs, we have defined several aspects that encapsulate specific testing concerns. Based on these aspects we defined different test focuses that are used to test the GUI of an automotive infotainment system. The input were test cases which were generated from a statechart. In the

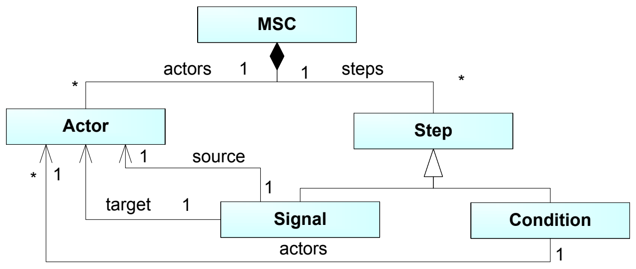


Fig. 7.18. A simple MSC metamodel.

following we demonstrate how it is possible to reuse the introduced aspects to instantiate an MSC based test case. Figure 7.17 shows such a test case for testing the communication behavior between phone application and GUI. The use case is the same as before: Incoming Call. The environment (in our case the test automation framework) calls the phone application. The application sends an incoming call message and a status message to the GUI via the communication bus. The GUI signals the incoming call, which is modeled by a condition. The environment sends an IDrive controller enter event, which should result in an accept call message from the GUI to the phone. For instantiating the MSC we want to reuse the IncomingCall aspect and the IDrive aspect to simulate the environment inputs. In order to achieve this, we have to adapt the existing pointcuts to the MSC metamodel. Figure 7.18 shows a simplified MSC metamodel. In a statechart, an environment input is defined by an *event*, which is defined in an MSC by a *signal*. As a consequence, we have to add another join point selection for *signal* in the pointcuts. To test if the correct messages are sent, we have to define a new aspect BusMessage that extends the BusService aspect. Listing 7.11 shows the extended pointcuts and the bus message aspect.

```
aspect IncomingCallTrigger {
  pointcut IncomingCall{
    Event : self.name = 'Incoming_Call'
    Signal : self.name = 'IncomingCall'
  }
  ...
}
aspect IDrive extends BusService{
  pointcut Enter{
    Event : self.name = 'Enter'
    Signal : self.name = 'Enter'
  }
  ...
}
aspect BusMessage extends BusService{
  pointcut BusSignal{
    Signal : self.source.name <> 'Tester'
  }
  advice MonitorBus before BusSignal <<
BusService.assertMessageSent("<%=signal.name%>")
  >>
}
```

Listing 7.11. Extended pointcuts.

7.6 Results

In Figure 7.5 we gave a short classification of test instantiation concerns. The case study showed that these concerns can easily be separated into different aspects using *AspectT*. This enabled us to reuse aspects in different test focuses. The possibility to inherit advice and pointcuts from other aspects is especially useful, because it enables the automatic inclusion of required aspects and their advice. Given a base of aspect definitions, the effort of test focus definition is reduced to selecting the required test oracles and test triggers. The required test framework configuration is automatically included based on aspect dependencies.

The idea of *AspectT* originated in the development of a test generation framework at BMW Group. The instantiation of generated test cases required a high configuration effort. By integrating *AspectT*, automatically generated test cases could be easily adapted to different testing contexts and could be applied in a broader range of testing scenarios.

A common problem is incomplete test or specification models. These models lack certain properties that should be tested or that are required for an automated execution of test cases. Using *AspectT* the missing parts could easily be integrated during instantiation which results in a greater area of application for specifications and test models.

Originally in the test framework, a developer had to implement the test script generation for each test focus. However, when applying the *AspectT* framework, the test script generation could be separated from the framework and test engineers could define their own *test focus* specific test script generation by using existing aspects and defining new aspects. This enabled the reuse of existing artifacts and allowed the test framework developer to focus on implementing test framework specific features. The test engineers did not have to rely on the framework developer for generating test scripts.

In addition, the results showed that test case generation was increasingly used in creating test preambles. The preamble in a test case establishes a certain precondition for the execution of the actual test. The precondition is defined by a pointcut, where the actual test is woven in the form of an advice and the aspect-based test generation is used to generate the preamble.

7.7 Related Work

Extensive research has been performed in the area of model-based testing. The main focus of these studies lie on behavior models, test selection criteria and algorithms. To the best of the author's knowledge, there are no studies on the requirements of model-based testing in different phases of system development and the consequences for test case instantiation. There are several case studies, where model-based test case generation has been evaluated for industrial case studies. In these case studies the test case instantiation is either performed by wrappers or by translation approaches. In [BL03], for example, a transformation approach that generates test scripts for test cases that are created from B [Abr96] specifications is presented. The test engineer defines a test script pattern and a mapping table. The tables contain the mapping between abstract model elements and their corresponding script patterns. Examples for wrapper approaches are the test case generation frameworks Torx [BFdV$^+$99] and TGV [JJ05]. Both provide the ability to execute the test cases during test generation time by using such wrappers around the SUT. In this case each test focus requires its own wrapper implementation. The combination of Torx or TGV with *AspectT* would combine the advantages of on-the-fly test case execution with the flexible test focus definition of *AspectT*.

Translation approaches are often based on a model-to-text generation framework. Several code generators for EMF models exist, such as JET[9] or XPAND2[10]. The main difference to these code generation frameworks is that, in *AspectT*, aspects can be arbitrary combined to a new code generator. In Jet, for example, a template must be defined for each *test focus*. All possible mappings between input model and target code are contained in this template. There is no modularization possible to reuse certain template parts in different code generators. In XPAND2 it is possible to divide a code generator into different templates. A template can expand other templates during its execution. Templates in XPAND2 can be reused in different code generators, but there must always be a base template that calls other templates depending on the input model. This is the primary difference to *AspectT*, where a template (the advice) defines where it is woven in the input model. XPAND2 also implements aspect-oriented concepts: each template can be extended by additional advice. The difference to *AspectT* is that the templates are join points rather than the input model. Therefore the join point model is at the wrong abstraction level for test case instantiation. On the other hand, these frameworks can have any kind of model as input whereas *AspectT* requires a model that has a logical tree structure. Another difference is the weaving: *AspectT* has a flexible weaving approach that allows the weaving during the creation of the input model.

The generation of source code from a tree structure is a common task in programming language compilers. Especially for attribute grammars [Knu68, Paa95, MJW00] many modularization approaches that divide an attribute grammar into separate modules exist, where each module can be reused in different programming language compilers to reduce the implementation effort for a new compiler. The advantage of *AspectT* and the test case instantiation is that only one concern has to be regarded in the modularization: the mapping between join point and code. This simplifies the modularization, in contrast to attribute grammars where more concerns such as production rules, attributes and semantic rules must be regarded as well.

[9] http://www.eclipse.org/modeling/m2t/?project=jet#jet
[10] http://www.openarchitectureware.org/

Gray et al. use aspect-oriented concepts to perform model evolution [GLZ06]. Their approach is based on a metamodeling framework similar to EMF: the Generic Modeling Environment (GME).[11] They define pointcuts using the Embedded Constraint Language (ECL) where a pointcut is defined as a query that selects a set of objects from a given model. An advice performs a model evolution for the objects selected by a pointcut. The model evolution is also defined using ECL. The query language in ECL is similar to OCL as used in *AspectT* where ECL includes additional constructs for model evolution.

Visser et al. introduced Stratego [Vis01]: a language for software transformation based on rewriting strategies. The basic idea of *AspectT* and Stratego are the same, transforming a tree structure into a textual representation. Stratego is a general-purpose program transformation language where *AspectT* focuses on test case instantiation. *AspectT*'s advantage is its simple notation and its intuitive way of describing a test trigger or test oracle by using concepts of AOP. Another difference is the dynamic weaving. Aspects can be woven while the input tree is dynamically created, which is important for runtime verification or simulation based test case generation.

The Motorola WEAVR developed by Cottenier et al. is an aspect-oriented modeling approach based on UML: composite-structure architecture diagrams and statecharts describe the structure and behavior of a system. WEAVR provides the ability to describe crosscutting concerns using pointcuts and advice. Pointcuts select transitions and actions. Advice are defined as statecharts. For statechart based test case generation this approach could be used to add missing constraint (e.g. timing) and missing input behavior. However, the mapping between woven statecharts and driver components cannot be described. WEAVR is restricted to statecharts where *AspectT* is modeling language independent.

7.8 Summary

Test case instantiation is an important part of model-based testing. Especially in embedded systems the abstraction gap between generated test cases and SUT is large, because it involves complex test setups that monitor and trigger the system. The test focus must be adapted to each possible test setup and test context. One way to minimize the effort of model-based testing is to optimize the test focus definition. The approach presented in this chapter uses aspect-orientation for separating different testing concerns. Each particular mapping between test model and test script is encapsulated in a separate aspect. This reduces the effort of test focus definition to the selection of aspects corresponding to the test goal. Another benefit of *AspectT* is that pointcuts are potential test selection criteria. This enables the combination of test focus definition and test case generation to generate test focus specific test cases.

We implemented our test focus definition approach in the language *AspectT*. The language is based on the EMF, where every EMF model is a potential join point model. This enables the application of *AspectT* to any modeling

[11] http://www.isis.vanderbilt.edu/projects/gme/

language, as long as the test case's metamodel is defined in EMF. The advice definition in *AspectT* implements a template based code generation approach that enables the generation of test scripts in any programming language. AspectT has been integrated into an existing test generation framework at BMW Group and has been successfully applied to testing an automotive infotainment system.

Part IV

Evaluation and Discussion

8

From Use Cases to Task Models

In the previous chapters, we introduced *TTask*, a task modeling notation that enables the modeling of features and their interactions. Furthermore, we presented the generation of test cases from *TTask* models based on new test selection criteria. This involved the refinement of task sequences with existing behavior models and the test script generation with *AspectT*. In the following three chapters we show that the application of our approach is feasible in a real world setting and that it enables the systematic test coverage of feature interactions. We show this by different case studies of real world examples from the automotive domain. Throughout the following three chapters we illustrate three points:

1. Show how to create a *TTask* model, based on an existing system specification (this Chapter).
2. Show the combination of task sequence generation, refinement, and test script generation for a running real world example (Chapter 9: "Case Study").
3. Show that the introduced test selection criteria cover critical scenarios (Chapter 10: "Evaluation of Test Selection Criteria").

This chapter focuses on the first objective: how to create a *TTask* model from a given specification. The chapter is structured as follows. In the first section we describe the translation of a real world use case specification into a *TTask* model. In the second section we compose these use case specific tasks to a task model that describes the temporal dependencies between the use cases.

8.1 Creating Tasks from Use Cases

In this case study we model the *software update functionality* of an infotainment system with *TTask*. This functionality allows the user to perform a software update of the infotainment system. The OEM provides software updates, which the user can save on a USB flash drive. After the user has connected the USB flash drive to the infotainment system, the update starts via the GUI. Furthermore, the functionality provides features to view installed updates and to uninstall previously installed updates. The functionality is specified in the form of use cases. Our approach is to systematically translate these use cases into a *TTask* model. The translation is performed in six steps:

1. Create a task for each use case.
2. For each step in the informal use case description, create a subtask in the corresponding task.
3. Model the temporal dependencies between the use case specific subtasks based on the use case description.
4. For each exception that is described in a use case, create a corresponding subtask. For each expected system reaction to the exception, create a subtask as well.
5. Model the dependencies between the tasks that describe the use case behavior and the tasks that describe the exceptional behavior.
6. Compose all use case specific tasks by modeling their temporal dependencies.

In the remainder of this section we introduce the use cases of the software update functionality and model them in *TTask*. In the following section we describe the composition of the use case specific tasks into a complete *TTask* model.

8.1.1 Use Case: Software Update Detection

The first use case in Table 8.1 describes the detection process of a USB stick. Figure 8.1 shows the corresponding task Detect USB Flash Drive. This is a composite task that implements the *selection pattern* with three subtasks. Each subtask describes a different USB flash drive detection scenario:

1. The USB flash drive contains no update data.

Use case:	Detection of software updates on USB flash drive
Description:	The customer connects the USB flash drive to the USB port in the car. The storage device contains an update file. The software update agent detects the update file. All detected updates are reported to the Software Update Manager and are forwarded to the GUI.
Prior Condition:	USB flash drive with appropriate update file is connected to the USB connector.
Post Condition:	The system has detected updates on the USB flash drive.
Initiator:	User
Exceptions:	The system should notify the user if the USB flash drive contains an invalid update.

Table 8.1. Use case: Software update detection.

2. The USB flash drive contains invalid update data.
3. The USB flash drive contains valid update data.

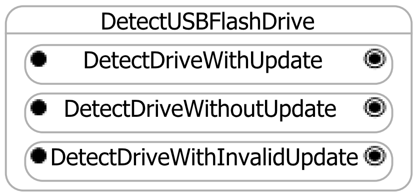


Fig. 8.1. Update detection task.

The subtasks have no temporal dependencies. Thus the task Detect USB Flash Drive is performed when one of its subtasks is performed. The second scenario is an example, how faulty inputs can be described in *TTask* by being encapsulated in separate tasks. This demonstrates how scenarios that involve faulty environment inputs can be covered in task-based test case generation.

8.1.2 Use Case: Display Installed Updates

The user can review previously installed updates. This part of the software update functionality describes the second use case in Table 8.2. Figure 8.2 shows the task Show Installed Updates which describes the use case. There is no exceptional behavior in this use case described. Thus the corresponding task has only two subtasks that model the sequential execution of the use case. This is a typical example for a GUI dialog, in which the user queries information from the system. The user performs the first task by requesting a list of previously installed updates from the system via the GUI. The second task describes the corresponding system reaction.

Use case:	Display currently installed updates and versions via GUI
Description:	The customer or the service partner determines installed updates using the GUI. 1. The customer/service partner selects the option “display installed updates”. 2. The currently installed software updates are displayed.
Prior Condition:	The system is idle.
Post Condition:	The system is idle.
Initiator:	User
Exceptions:	

Table 8.2. Use case: Display software updates.

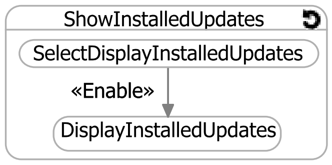


Fig. 8.2. Display updates task.

8.1.3 Use Case: Software Update Removal

If the user is not satisfied by the previously installed update, it can be removed from the system. The third use case in Table 8.3 describes the uninstallation procedure which removes the last installed software update. The use case is modeled by the task in Figure 8.3. This use case also describes a sequential dialog between user and system. The system notifies the user over the uninstallation progress, which is modeled by the *long running task* Show Uninstallation Progress.

Use case:	Uninstallation of the software update
Description:	The last software update that has been installed is uninstalled by the customer using the update assistant. 1. The customer selects the software update uninstall function. 2. The customer is informed about the uninstall progress. 3. Visible confirmation of successful uninstall is given.
Prior Condition:	There are installed updates.
Post Condition:	
Initiator:	User
Exceptions:	

Table 8.3. Use case: Uninstall software updates.

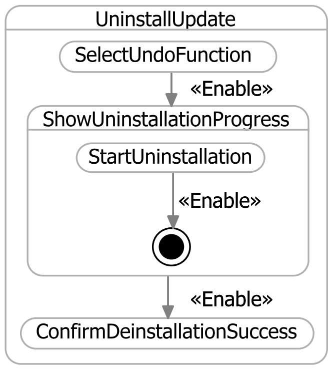


Fig. 8.3. Uninstall update task.

8.1.4 Use Case: Software Update Installation

The fourth use case in Table 8.4 describes the installation of a software update. The corresponding task in Figure 8.4 is divided into two subtasks. The first subtask Perform Update describes the normal update procedure, as specified in the use case. However, the user can remove the USB flash drive during the update procedure. This is a typical example of exceptional behavior that must be tested during development. In particular, it is important to cover every possible situation in which the exception might occur, in order to ensure a stable state in every scenario. The second subtask Handle Removal Exception models the situation when the user removes the USB flash drive during the update procedure. The task Handle Removal Exception is only *enabled* when the task Perform Update is *active*. When the task Confirm Successful Installation stops, the task Handle Removal Exception becomes *disabled*. However, the update procedure is disabled when the USB drive is removed during the update. This is described by the task Remove Drive When Update Active and its *disable* dependency to the task Perform Update. The system's reaction to the exception models the task Show Update Failed which is *enabled* by the task Remove Drive When Update Active.

Use case:	Installation of a software update
Description:	The customer or the service partner determines installed updates using the GUI. 1. The customer follows the instructions on the GUI and confirms the installation of the update. 2. The customer is informed about the progress of installation. 3. Visible confirmation of successful installation is given.
Prior Condition:	The system has detected new updates on the USB flash drive.
Post Condition:	The system has detected updates on the USB flash drive.
Initiator:	User
Exceptions:	If the USB flash drive is removed during installation, the system should undo the update and notify the user.

Table 8.4. Use case: Software update installation.

8.2 Composing the Use Case Tasks

The previous section described each use case of the software update functionality in *TTask*. In order to describe the complete software update functionality, we have to model the dependencies between these use cases. Each use case represents a specific feature. Hence, the interactions between these use cases are feature interactions. We define these intentional feature interactions using task dependencies. Figure 8.5 shows the complete task model for the software update functionality. The subtasks that have no new dependencies are hidden for the sake of conciseness. First of all, we have to add another task Remove USB Flash Drive in order to describe the non-exceptional removal of a USB drive. The normal removal of a USB flash drive has not been described in the use case specification. The incompleteness of the specification became obvious when we simulated our task model.

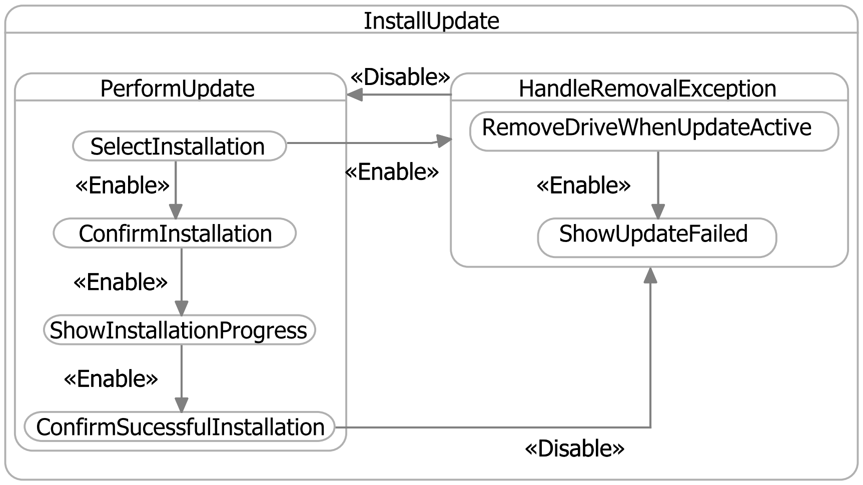


Fig. 8.4. Update installation task.

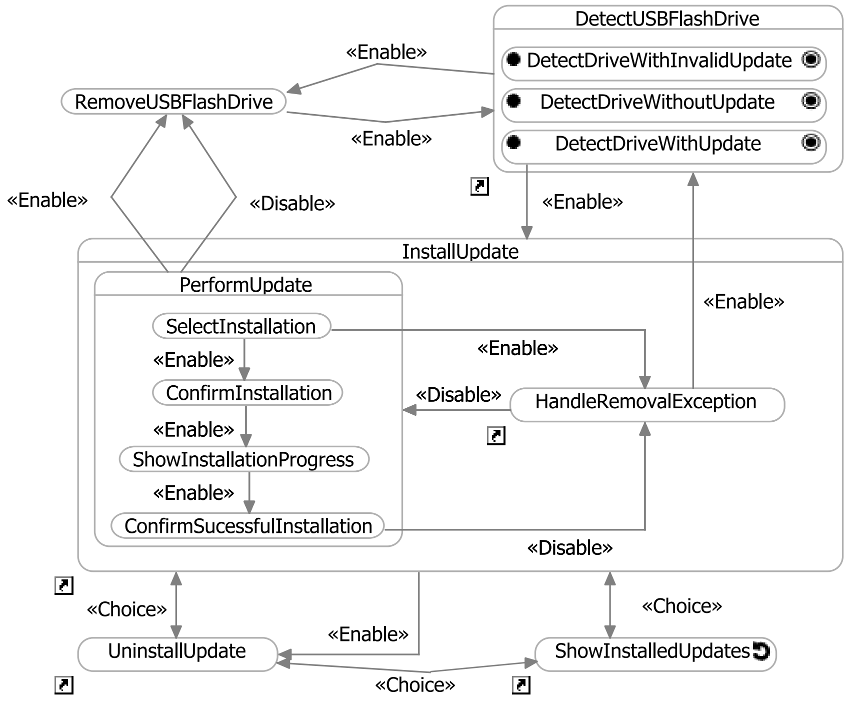


Fig. 8.5. Software update taskmodel.

There are three reasons for feature interaction:

1. **Shared usage of the GUI:** The three tasks Install Update, Show Installed Updates, and Uninstall Update are controlled via the GUI. Hence, they are all mutually exclusive because the GUI supports only the execution of one task at a time. This is modeled by the *mutual exclusion pattern*, which is symbolized by the choice dependency.

2. **Required USB Flash Drive:** The task Install Update requires a valid update on a USB flash drive. This is modeled by the *enable* dependency between Detect Drive with Update and Install Update. However, there are two tasks for the removal of a USB drive. One that represents the normal removal of a USB drive and one that represents the exceptional removal during the update procedure. The exceptional removal is only possible during the update procedure. Hence, it is necessary to ensure that when the update is started, only the exceptional task is enabled. This is modeled by the *disable* and *enable* dependency between Perform Update and Remove USB Flash Drive. Furthermore, when the task Handle Removal Exception is performed, it disables Perform Update and enables Detect USB Flash Drive.

3. **Installed Updates:** The task Uninstall Update can only be performed if an update has already been installed. This is modeled by the *enable* dependency between Install Update and Uninstall Update. However, this is an example for a situation in which the current semantics of *TTask* do not suffice. The scenario, in which the user performs two updates in a row and then uninstalls both updates consecutively, cannot be modeled with *TTask*. This would require the possibility to model additional preconditions and actions in a task, based on a common data model, for example, a counter variable, which is currently not supported in *TTask*.

After adding the temporal dependencies between the use case specific tasks, the task model describes all usage scenarios of the *software update functionality*.

8.3 Summary

This chapter presented the first of three case studies of task-based test case generation with *TTask*. This chapter focused on the first step in task-based test case generation: modeling features and their interactions with *TTask*. To demonstrate *TTask* we created a *TTask* model from an existing use case specification of the *software update functionality*. This chapter showed that the transformation of an informal use case description into a *TTask* model is straightforward. Furthermore, we have seen how *TTask* can be used to model exceptional behavior as well as temporal dependencies between different use cases. In summary, this chapter showed that the effort of creating a *TTask* model from a given specification is reasonable. Such a *TTask* model enables task sequence generation in order to systematically cover the specified behavior. The generation of task sequences and the subsequent generation of executable test scripts is the topic of the following two chapters.

9

Generating Test Cases with *TTask*

This chapter presents a case study in which we apply task-based test case generation to a real world example. The first objective of this study is to demonstrate our approach of task-based test case generation and test instantiation by a running example that involves all necessary steps to create executable test scripts from a task model. The second objective is to show the feasibility of our approach for a real world example from the automotive domain. In order to fulfill the objectives, the case study comprises all steps of task-based test case generation:

1. Modeling features and their interactions in *TTask*, which has been introduced in Chapter 4.
2. Generation of task sequences based on test selection criteria, which were introduced in Chapter 5.
3. Refinement of the generated task sequences using existing behavior models, following the approach described in Chapter 6.
4. Generation of executable test scripts with *AspectT* (see Chapter 7).

This chapter is structured as follows. The first section introduces the environment at BMW Group in which the case study has been performed. The second section introduces the features of a real world infotainment system that are modeled using *TTask*. The third section presents the task sequence generation from the task model. The fourth section demonstrates test case instantiation with *AspectT*.

9.1 Environment

This section introduces the environment in which this case study has been performed. This includes the test setup consisting of the SUT and the test execution environment. Model-based testing strongly depends on tools that support the process of test case generation and execution. We created such a tool for *TTask*: *Task-GEN*. *Task-GEN* is a modeling environment and test case generator for task models.

9.1.1 Test Setup

The targeted system comprises all ECUs of the infotainment system due to the regarded features. For testing, these ECUs are integrated in a test rack where they are connected via the MOST and the CAN busses. Test scripts are executed from a test host that is connected to the MOST and CAN networks. In order to receive and make phone calls, the AIS is connected to a mobile phone via Bluetooth. In order to stimulate and receive these phone calls, the test host is connected to another mobile phone. Figure 9.1 depicts the testing environment.

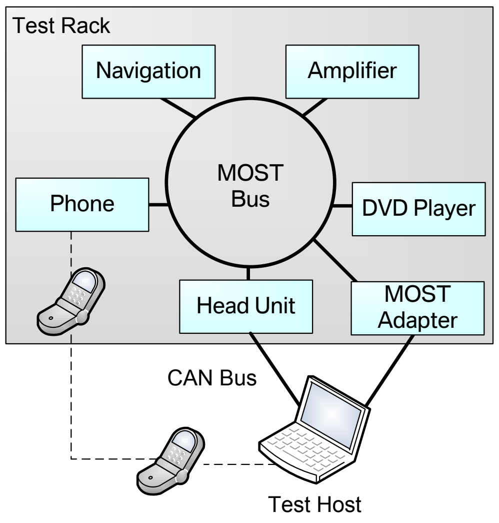


Fig. 9.1. Testing environment.

9.1.2 Test Execution Environment

Model-based test case generation results in a potentially large number of test cases. In order to execute such a large number of test cases, the execution of them must be automated. However, the automated execution of test cases requires an environment that enables the automated stimulation of the SUT as well as automated test oracles. At BMW Group, test cases for infotainment systems are written in Python[1] and use the BMW Test Automation Framework (TAF). TAF provides different test drivers which encapsulate functionalities to trigger and observe the system. An example is the ZBEService[2] that provides a simple application programming interface (API) to trigger joystick inputs via bus messages:

```
ZBE.Right() # send joystick turn right signal
ZBE.Press() # send joystick press signal
```

[1] http://www.python.org
[2] ZBE stands for "zentrales Bedienelement" which is German for central control unit.

Test oracles are defined using test drivers that observe the system. For example, the following test oracle checks whether the screen change is fast enough:

```
MMI.WaitForText("Enter Destination", 1000)
except TimeOut:
   log(ERROR, "Timeout enter destination")
```

The MMIService enables the monitoring of the GUI using a screen grabber and OCR. The call to the WaitForText functions waits until the specified text occurs in the cursor or a timeout occurs. If a timeout occurs, a corresponding error is logged.

9.1.3 Task Modeling Environment - *Task-GEN*

Task-GEN is a graphical modeling environment for *TTask* models. A modeler can use *Task-GEN* to create task models with the graphical syntax that is described in Chapter 4.3. Furthermore with *Task-GEN* the modeler can define multiple views on the same task model. This is advantageous for models with a large number of tasks, as the tasks can be distributed across multiple diagrams. The concept of different views on the same task model is depicted in Figure 9.2. This enables the separation of feature-specific behavior from feature interactions. We use the following task modeling procedure for our case study. First, all features are described in separate views. Then, the interactions between these features are described in additional views.

Task-GEN can automatically transform task models into *Promela*. There are two different modes for the transformation: one for the simulation of *TTask* models, and one for test case generation from *TTask* models. In order to simulate a *TTask* model, *Task-GEN* creates an open *Promela* model in which the start and stop of a task can be triggered by the modeler. The modeler can simulate the execution of tasks and *Task-GEN* provides visual feedback about the current task model state. Figure 9.3 shows a screen shot from *Task-GEN* in simulation mode. The left part of the screen shot shows the simulated task model. The task modes are represented by different colors during simulation. The right part shows the control view where the modeler can start and stop

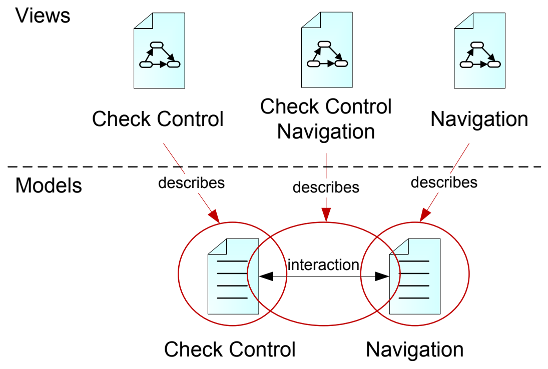


Fig. 9.2. View concept.

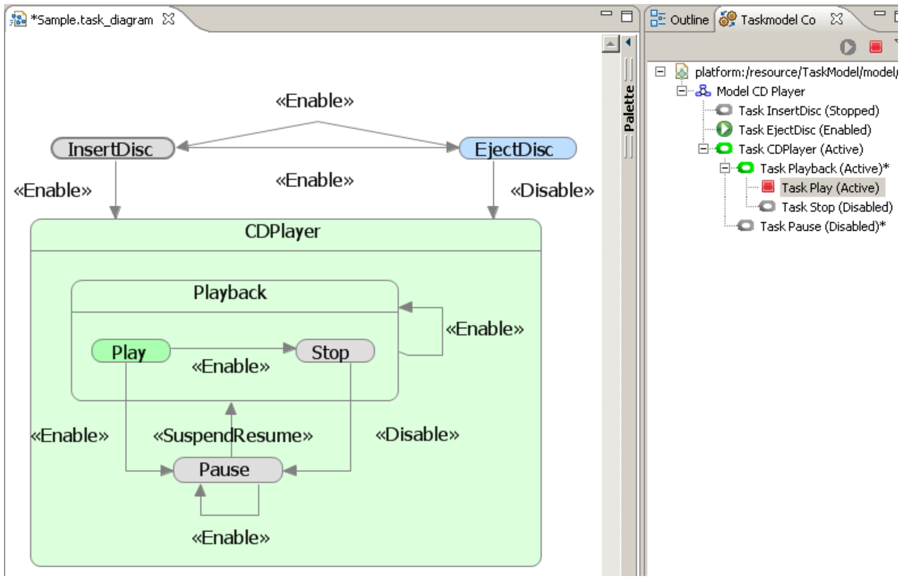


Fig. 9.3. *Task-GEN*.

the execution of a task. The actual simulation of the task model is performed with SPIN which runs in the background.

In order to generate test cases from a task model, *Task-GEN* supports the generation of closed *Promela* models. Based on these closed *Promela* models, *Task-GEN* can generate task sequences using the test selection criteria that were introduced in Chapter 5.4. Test case generation is performed in multiple steps. These steps are defined in the form of openArchitectureWare (OAW)[3] workflows.

Figure 9.4 shows the complete workflow for the generation of test scripts from a task model. We use *TTask* to depict the workflow. Each task represents a step in the workflow. However, the actual workflow is specified in XML. First, a task model is loaded and transformed into *Promela* code. The *Promela* code is then compiled into an executable model checker program. In the next step, a set of LTL formulas are generated for a given test selection criterion. Each LTL formula describes a trap property. For each generated LTL formula a task sequence is generated. This is performed by using SPIN to check the LTL formula against the model checker program. When SPIN finds no counter-example for the trap property, the task sequence generation is finished. Otherwise, SPIN creates a trail for the counter-example. *Task-GEN* parses the trail file and transforms it into a task sequence model. This task sequence model is then transformed into test scripts with *AspectT*. For each test focus, *AspectT* is invoked by loading a test focus specific aspect setup and by weaving it into the generated task sequence.

In the remainder of this chapter, we describe this workflow on the basis of a real world example.

[3] http://www.openarchitectureware.org

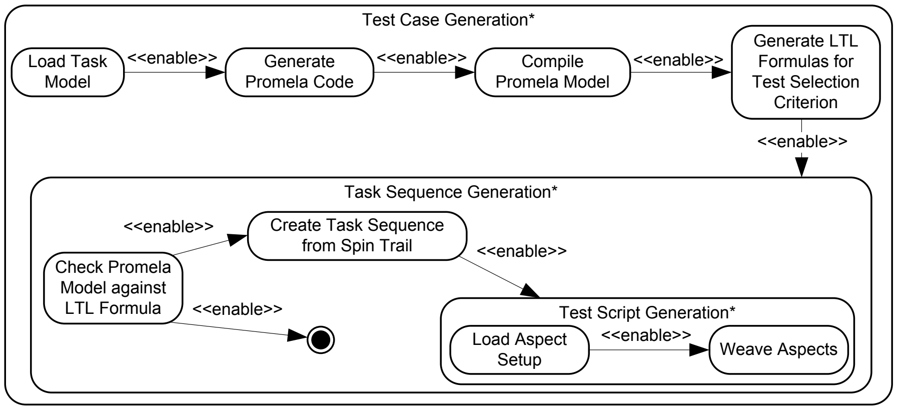


Fig. 9.4. Test script generation process.

9.2 The Task Model

A real world infotainment system integrates more than 1000 different features. The case study comprises a representative subset of these features. The features are Enter Destination, Route Guidance, Change Route Options, Listen to Radio, Incoming Call, Start Call and Check Control Messages. In the first part of this section, we describe the feature-specific tasks and in the second part we describe the interactions between these features by their task dependencies.

9.2.1 Modeling the Features

The first step is to model each feature by a corresponding task. Then, these tasks are subsequently refined into different subtasks that describe the feature-specific behavior. In the following, we describe the features that comprise the case study.

Check Control:

The check control function in a car monitors the vehicle and notifies the driver when an exceptional condition occurs. This involves also critical situations, such as when the tire pressure is below a critical value. When the check control function observes such a critical condition it gives an audible warning signal and notifies the GUI. The GUI subsequently shows a warning screen that interrupts all other active dialogs. The driver must confirm that the warning has been noticed in order to move the warning screen into the background.

Figure 9.5 shows the task model for the check control message Handle Low Tire Pressure. First the system task is modeled that observes a low tire pressure. This task enables the Handle Low Pressure task which contains tasks for the warning signal and the warning message screen.

Radio:

An AIS supports different entertainment features, for example, the reception of television (TV) and radio and the playback of CDs and MP3s. We exemplify

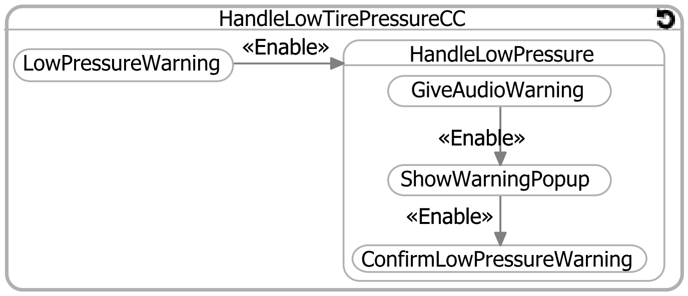


Fig. 9.5. Check Control Message.

these features by integrating the radio functionality in our case study. In this study we focus only on one part of the radio feature, the playback of a radio station after the selection of a radio station. Figure 9.6 shows the task Listen to Radio that models the radio feature. The task Listen to Radio uses the *long running task* pattern to model its ability to perform in an interleaved fashion with other tasks. The task Listen to Radio is initiated by the task Select Station which is performed when one of its subtasks is performed. The selection of different radio stations is modeled using the *selection pattern*.

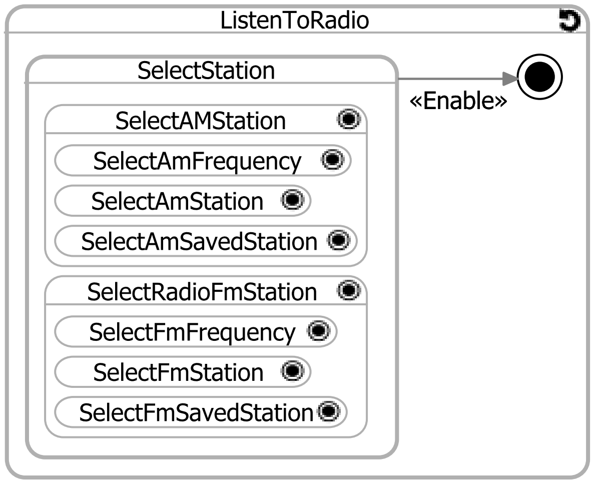


Fig. 9.6. Radio.

Navigation:

One of the most important functionalities of an AIS is the navigation system, and it is one of the most complex applications as well. In this case study we model the basic navigation features destination input and route guidance. We describe the navigation functionality with three tasks: Enter Destination, Guidance and Change Route Criteria. The Enter Destination task has four subtasks for the selection of the country, city, street and house number. In the navigation feature, the country is initially set to the user's country. Hence, the user can start with either Enter Country or Enter City. This is modeled by setting the initial mode of Enter City to enabled (indicated by the black circle on the left side of the task). The user can always change a selected route and therefore the task Enter Destination is never stopped once it is started. However, when the user changes the country, the tasks Enter House Number and Enter Street get disabled. When a city is selected, the route guidance can be started, which is modeled by the *enable* dependency from the task Enter City to the task Start Guidance.

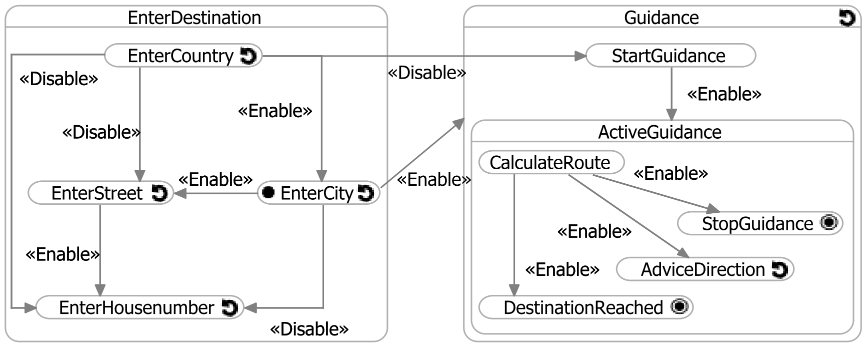


Fig. 9.7. Navigation.

Telephony:

The user can integrate a mobile phone into the AIS via Bluetooth. In this case study we regard the two main features Make Phone Call and Incoming Call. Figure 9.8 shows the task model that describes these features. A call can either be started by entering a number or by selecting a contact. When a call is started, the system dials the number and waits for the callee to accept the call. The user can cancel the dialing process, which results in the end of the Start Call task. When the Dialing task is finished, the callee can either reject or accept the call. If accepted, the task Call Speak becomes active, otherwise the user can set an active call on hold which suspends the call. An active call is ended either by the user or by the callee.

The task Incoming Call in Figure 9.9 describes the incoming call handling. This task is started when the system performs the task Signal Incoming Call. The system signals an incoming call by showing a popup dialog in the GUI and by playing an audio signal. The call handling is similar to the task Make Phone Call.

9.2.2 Modeling the Feature Interactions

The previous section described the features and their behavior with tasks. The interactions between these features are not specified yet. In this section, we discuss two main intentional feature interactions between the introduced features. The first is the shared usage of the GUI. The second is the audio channel that can only be used by one feature at a time.

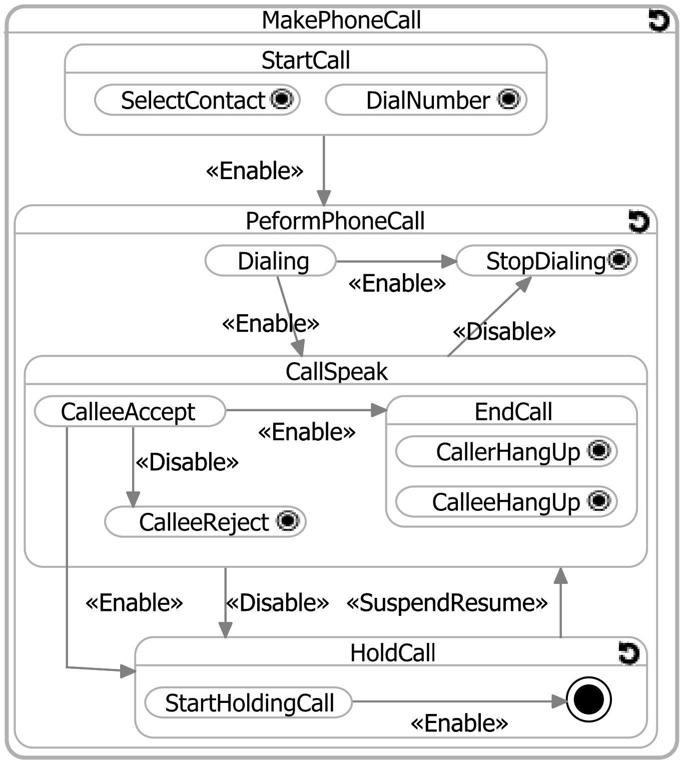


Fig. 9.8. Call.

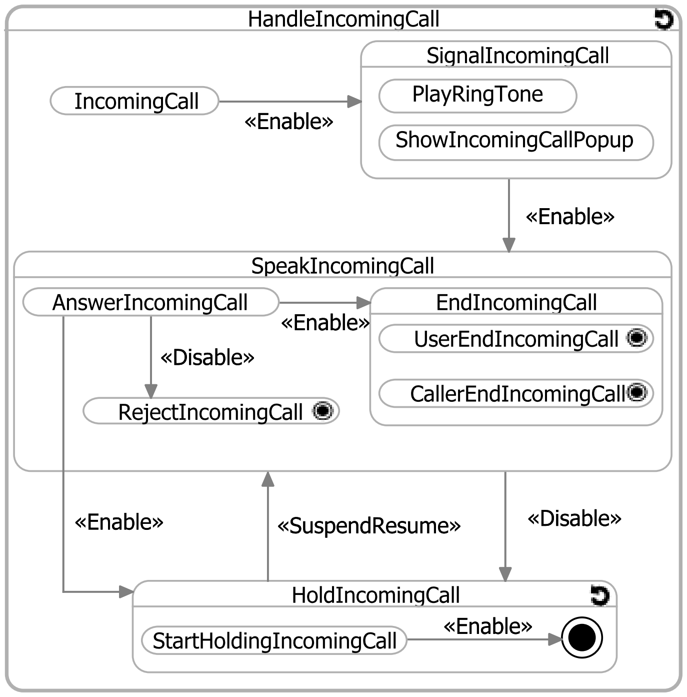


Fig. 9.9. Incoming Call.

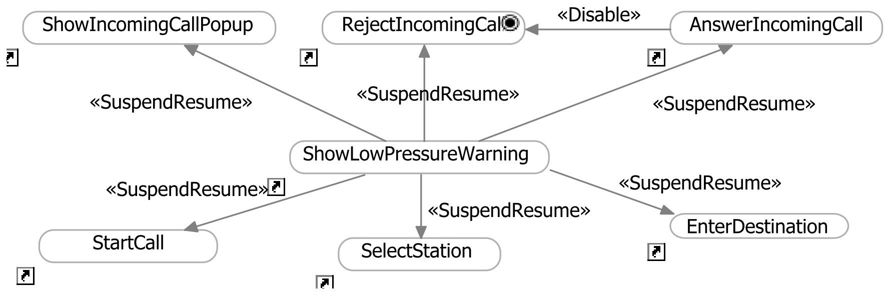


Fig. 9.10. GUI-based feature interactions.

GUI-based feature interactions:

Figure 9.10 shows the intentional feature interactions that result from the shared usage of the GUI. The low tire pressure warning popup suspends all other GUI dialogs until the user confirms the warning. Hence, the task ShowLowPressureWarning suspends all other tasks that involve the GUI, such as Select Station or Enter Destination. All other tasks that involve the GUI can be performed interleaved.

Audio-channel-based feature interactions:

Figure 9.11 shows the intentional feature interactions that results from the shared usage of the audio device. On the one hand there are long running tasks that use the audio channel, such as Listen to Radio, Perform Call, and Handle Incoming Call. On the other hand there are atomic tasks that use the audio channel and suspend the long running tasks, for example, Advice Direction and Give Audio Warning. Furthermore, an active phone call is interrupted by an incoming call, which is modeled by the *suspend* dependency between Handle Incoming Call and Make Phone Call, as shown in Figure 9.12.

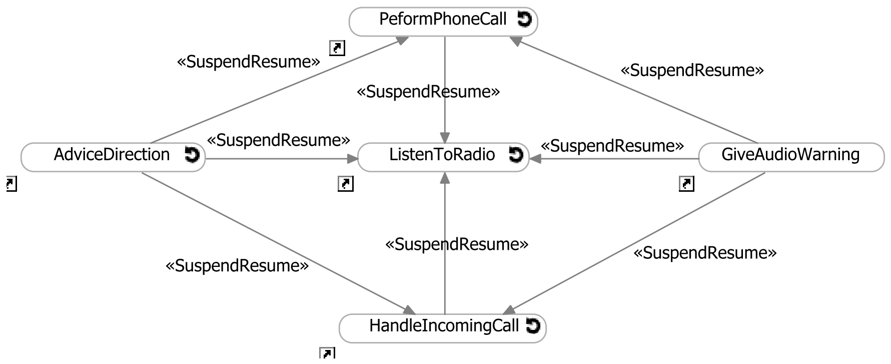


Fig. 9.11. Audio-channel-based feature interactions.

The previous three task model views describe the interactions between the features of our study. The composition of all task model views describes the task model that we want to use for test case generation. The next section introduces the task sequence generation from this task model.

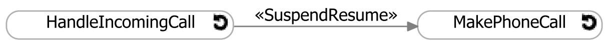


Fig. 9.12. Incoming call suspends Phone Call.

9.3 Generating Task Sequences

Chapter 5.4 introduced several coverage-based test selection criteria. This section describes the application of these test selection criteria to the task model introduced in the previous section. During our case study we generated first the corresponding LTL formulas for each test selection criterion and then we used SPIN to produce counter-examples for each LTL formula. SPIN creates for a counter-example a trail file that describes the path in the *Promela* model that leads to a violation of the given LTL formula. *Task-GEN* parses these trail files and creates a task sequence. Figure 9.13 shows one of the generated task sequences in the form of an MSC.

The task model that we used in this case study comprises 60 tasks where 38 tasks are atomic tasks. Figure 9.14 shows the number of generated task sequences for each test selection criterion. The chart shows that the *concurrency coverage criterion* generates the most task sequences. This is no surprise because all possible combinations of concurrent task executions are covered. Thus the testing effort is particularly high, when all scenarios should be covered in which unintentional feature interferences might occur.

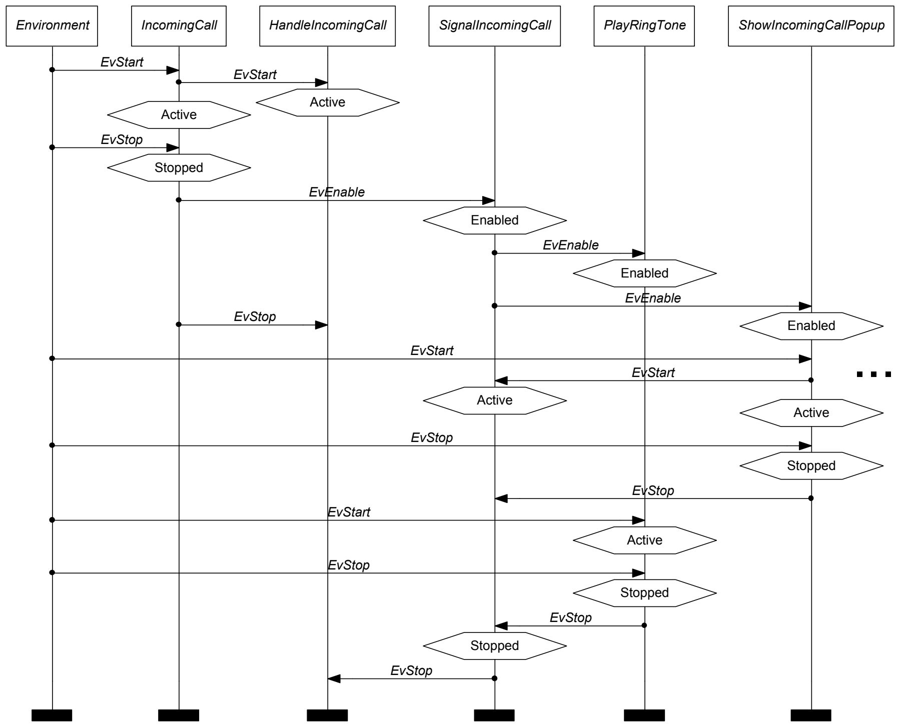


Fig. 9.13. Task sequence example.

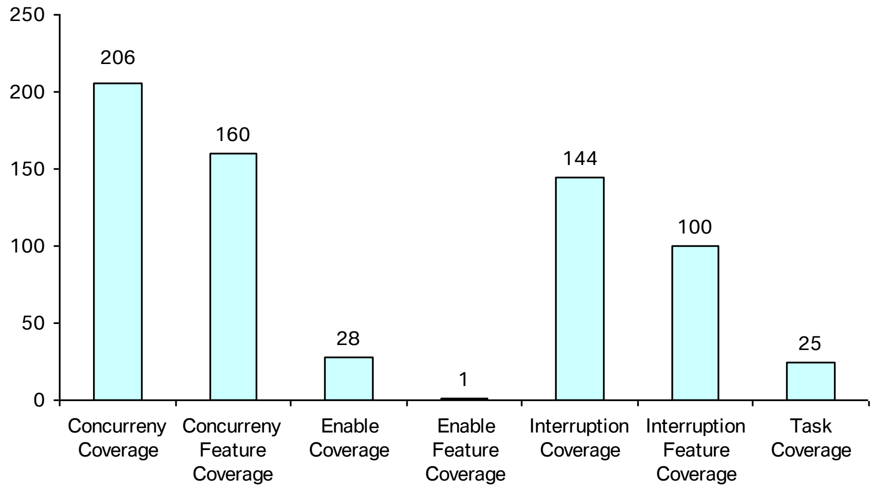


Fig. 9.14. Number of generated task sequences.

The *interruption coverage criterion* generated a large number of test cases as well. This depends partly on the domain, because in our task model most features interrupt each other. Furthermore, one *suspend* dependency stands for multiple suspension scenarios. For example, the task Enter Destination can be interrupted after each of its subtasks has been performed.

By contrast, the *enable coverage criterion* generated a relatively small number of task sequences. This depends on the task model in which only one *enable* dependency exists between different features. Furthermore, *enable* dependencies were mostly defined between atomic tasks. Thus every *enable* dependency is covered by one task sequence.

The number of task sequences generated by the *task coverage criterion* is even lower than the number of tasks in the task model. This is due to only generating trap properties for tasks that have not already been covered by one of the generated task sequences.

The next chart in Figure 9.15 compares the average length of the different task sequences. The length of a task sequence is determined by the number of executed atomic tasks. Our results show that the lengths of the task sequences are reasonably small. The increased task sequence length of the interruption and concurrency coverage criteria results from the more complex scenarios they cover; usually, one task must be started in order to be interrupted or interleaved by another task.

This section demonstrated the generation of task sequences from a task model. Our example shows that use case specific test selection criteria, such as task coverage or enable coverage, produce a relatively small number of task sequences. However, as soon as unintentional feature interaction scenarios are included into the task sequence selection process, the number of generated task sequences increases significantly. These results correspond to the ones from other case studies that we performed.

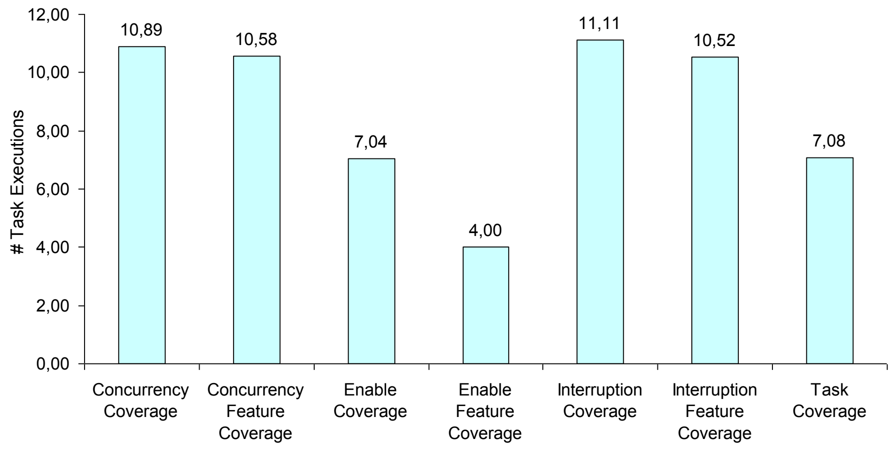


Fig. 9.15. Average task sequence length.

The next step is to enrich these task sequences with input and output behavior in order to transform them into test cases. This is covered in the next section.

9.4 Task Sequence Refinement and Instantiation

In this section we describe the transformation of the generated task sequences into executable test scripts with *AspectT*. Furthermore, we show how *AspectT* can be used to combine test case refinement and test case instantiation.

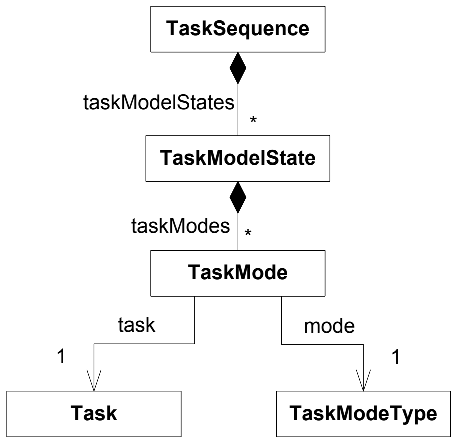


Fig. 9.16. Task sequence metamodel.

SPIN generates a trail file that describes a specific task sequence. *Task-GEN* parses this trail file and creates an EMF task sequence model. Figure 9.16 shows the EMF metamodel of a task sequence. A task sequence consists of multiple task model states. Each task model state consists of multiple task

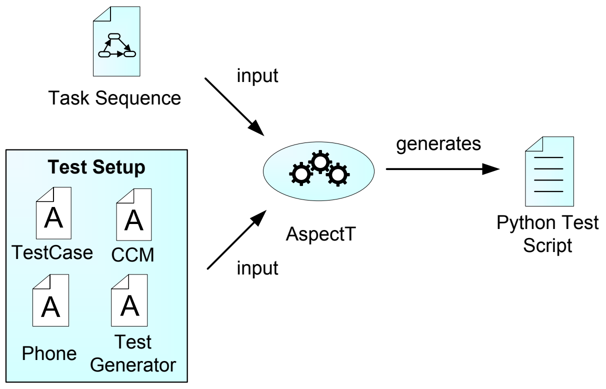


Fig. 9.17. Test focus.

modes. This EMF task sequence is input for the *AspectT* weaver which creates a python script for a given test focus.

The general approach is to encapsulate configuration code, test triggers, and test oracles into separate aspects. The configuration code is weaved before and after a task sequence in order to initialize and shutdown required services. Test triggers and test oracles are weaved at specific task modes. For example, the environment inputs to simulate the low tire pressure warning are weaved when the task Low Pressure Warning is *active*. In general, for each task that is started by the environment, a corresponding test trigger is defined. Tasks that are performed by the system, such as *Show Low Pressure Warning*, do not require a test trigger. Rather, they require a test oracle that tests whether the system performs the task correctly.

Configuration aspects, test trigger aspects, and test oracle aspects are composed into a test focus. Figure 9.17 shows the test focus that is introduced in this section. The test focus comprises four aspects: TestCase,TestGenerator, CheckControlMessage and Phone. The aspects CheckControlMessage and Phone encapsulate the test triggers and oracles for the corresponding features and the aspects TestCase and TestGenerator encapsulate general initialization code.

Listing 9.1 shows the implementation of the aspect TestCase which contains the python class header and the method declaration that starts the test script. TestCase comprises a pointcut that selects all task sequences and an advice that weaves the header code *before* an object of the type *task sequence*.

```
aspect TestCase {
  pointcut TaskSequence {
    TaskSequence
  }

  advice Header base TaskSequence
  <<
from mmi.base.MMITestBase import MMITestBase

class <%=self.name %>(MMITestBase):

  def __init__(self, methodName='runTest'):
    MMITestBase.__init__(self, methodName, '<%=self.name %>')

  def runTest(self):
    print 'Starting test <%=self.name %>...'
  >>
  }
```

Listing 9.1. Test case header.

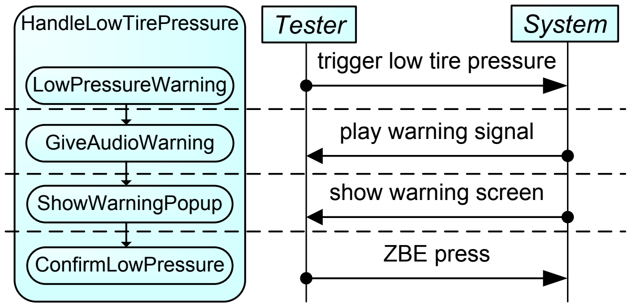


Fig. 9.18. Handle CC message scenario.

The next aspect in Listing 9.2 demonstrates the scenario-based refinement of the task Handle Low Tire Pressure CC with *AspectT*. Figure 9.18 shows the scenario that describes the task sequence Low Pressure Warning → Give Audio Warning Show Warning Popup → Confirm Low Pressure Warning. Therefore, we have to define two test triggers for the test inputs and two test oracles for the system reactions in *AspectT*. The aspect CheckControlMessage in Listing 9.2 implements the corresponding test triggers and test oracles. There are two pointcuts, one that selects the *TaskMode* where LowPressureWarning is *active* and one that selects the *TaskMode* where ConfirmLowPressure is *active*. There are three advice implemented in the aspect CMM. The advice LowPressureWarning_Trigger implements the test trigger that sends a check control (CC) message via the CAN bus. The test oracle is implemented in TirePressureLow_Oracle. The oracle implemented in the advice waits for the CC message screen to show and logs an error if the screen is not shown within one second. The confirm message trigger is implemented in the advice ConfirmMessage_Trigger by simulating a press by the user.

```
aspect CheckControlMessage {
  pointcut LowPressureWarning {
    TaskMode : self.state = Active and
                    self.task.name = 'LowPressureWarning'
  }
  pointcut GiveAudioWarning {
    TaskMode : self.state = Active and
                    self.task.name = 'GiveAudioWarning'
  }
  pointcut ShowWarningPopup {
    TaskMode : self.state = Active and
                    self.task.name = 'ShowWarningPopup'
  }
  pointcut ConfirmLowPressure {
```

```
  TaskMode : self.state = Active and
                  self.task.name = 'ConfirmLowPressure'
}

advice LowPressureWarning_Trigger base TirePressureLowTask
<<
  CAN.CCM(1, 1) # trigger \gls{cc} message  >>

advice GiveAudioWarning_Oracle before GiveAudioWarning
<<
  AUDIO.waitForGong(2)
  except TimeOut:
    log(ERROR, "Low Tire Pressure warning signal timeout")  >>

advice ShowWarningPopup_Oracle base ShowWarningPopup
<<
  MMI.waitForText("Low Tire Pressure", 2)
  except TimeOut:
    log(ERROR, "Low Tire Pressure popup timeout")  >>

advice ConfirmLowPressure_Trigger base ConfirmLowPressure
<<
  ZBE.Press()  >>
}
```

Listing 9.2. CheckControlMessage Aspect.

Previously we have shown how scenarios are implemented with *AspectT*. In the following we introduce the state-machine-based refinement of a task model with *AspectT*. During component testing of the infotainment system, a specific test model already has been created to systematically test the GUI. The test model describes the focus behavior of the GUI in the form of a statechart. It is used to generate GUI specific test cases, such as text tests. Figure 9.19 shows a part of this model that describes the behavior of the navigation menu. We use this focus model for the refinement of our task models and thereby demonstrate the reuse of existing component models.

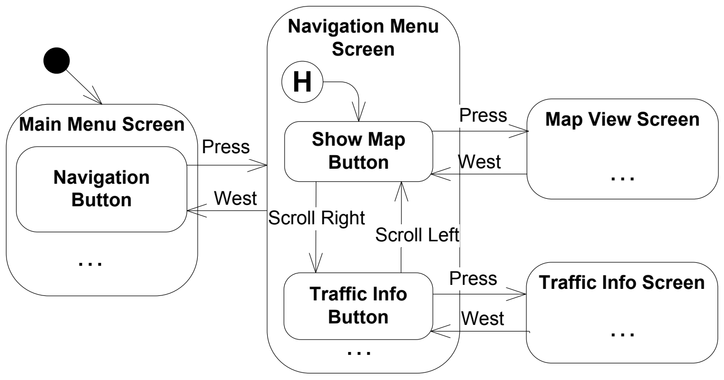


Fig. 9.19. Extract of the GUI focus model.

In order to generate test cases from the dialog model we use an existing test case generator that enables the generation of a shortest path between two states. The generator is implemented in Java and therefore can be called from within an advice in *AspectT*. The API of the generator is shown in Listing 9.3.

```
def testGenerator = new TestGenerator()
// load the statechart
testGenerator.init("model/spec/MMI_Model.mbt")
def result = testGenerator.generatePathTo("Enter_Destination_Speller")
// result = [ZBE.Right, ZBE.Right, ZBE.Press,...]
result = testGenerator.generatePathTo("Change_Language_Settings")
// result = [ZBE.West, ZBE.West, ZBE.Left,...]
```

Listing 9.3. Test generation.

The generator is initialized with a statechart model that represents the focus behavior of the GUI. After initialization, the generator can be used to generate inputs that establish a specific state in the GUI. In Listing 9.3 the first call:

```
testGenerator.generatePathTo("Enter_Destination_Speller")
```

generates the test inputs that establish the state Enter_Destination_Speller starting from the statechart's initial state. The second call:

```
testGenerator.generatePathTo("Change_Language_Settings")
```

generates the test inputs that establish the state Change_Language_Settings starting from the active state Enter_Destination_Speller.

Our approach is to encapsulate the test generator in the aspect TestGeneratorAspect which is shown in Listing 9.4. The aspect has an intertype declaration that contains the test generator. The generator is initialized in the advice InitTestGenerator which is invoked before a task sequence.

```
aspect TestGenerator {
  def testGenerator = <% new TestGenerator()%>

  pointcut TaskSequence {
    TaskSequence
  }

  advice InitTestGenerator before TaskSequence
  <<
    <% testGenerator.init("model/spec/MMI_Model.mbt") %>
  >>
}
```

Listing 9.4. Test generator initialization.

Each aspect that requires the test generator can extend the TestGenerator aspect and access the intertype variable to generate test inputs. As a consequence each aspect implicitly incorporates the task execution history when it generates an input sequence for the GUI. For example, the aspect that implements the task Select Contact uses the test generator to generate the input sequence that is necessary to navigate to the contact list. Listing 9.5 shows the implementation of the aspect. The test generator calls are implemented in the advice SelectContact_Trigger.

```
aspect Phone extends TestGenerator {

pointcut SelectContactActive {
    TaskMode : self.state = Active and
                 self.task.name = 'SelectContact'
  }

  ...

  advice SelectContact_Trigger base SelectContactActive <<
    # Navigate to SelectContact
<%
    def result = rteHelper.generatePath("C503_496")
    for (step in result) { %>
    <%=step%>
<%} %>

    # wait for contacts in cursor
    MMI.WaitForText("A A", 60)

    # Call contact
    ZBE.Press()
  >>
  ...

}
```

Listing 9.5. Call contact.

Furthermore, *AspectT* can be used to implement additional test oracles. For example, when the task Listen to Radio is suspended, the radio application should be muted. The aspect Radio in Listing 9.6 shows the implementation of a corresponding test oracle. The pointcut RadioSuspended selects any task model state where the task Listen to Radio is suspended. The advice RadioSuspended_Oracle contains the implementation of the test oracle that tests if the radio is muted.

```
aspect Radio extends TestGenerator {

pointcut RadioSuspended {
    TaskMode : self.state = Suspended and
                 self.task.name = 'ListenToRadio'
  }

  ...

  advice RadioSuspended_Oracle base RadioSuspended <<
    # Test if radio is suspended
    ...
  >>
  ...

}
```

Listing 9.6. Radio test oracle.

Resulting Test Scripts

Listing 9.7 shows the resulting test script for the task sequence which is shown in Figure 9.13. This script can be executed from a computer that is connected to a test bench by the CAN and MOST bus.

```
class TaskSelection000(MMITestBase):
  def __init__(self, methodName='runTest'):
    MMITestBase.__init__(self, methodName, 'TaskSelection000')

  def runTest(self):
    # Navigate to DialNumber
    ZBEL6.Right(1)
    ZBEL6.Right(1)
    ZBEL6.Press(1)
    # Dial Number via Speller
    MobilePhone.DialNumber("0179xxxxxxx")
    # Start call
    ZBEL6.Press(1)
    # Wait until dialing is finished
    time.sleep(10)
    # Callee Accept
    MobilePhoneControl.Answer()
    # Speak for 5s
    time.sleep(5)
    # trigger CC message
    CAN.CCM(1, 1)
    # wait for warning signal
    AUDIO.waitForGong(2)
    except TimeOut:
      log(ERROR, "Low Tire Pressure warning signal timeout")
    # wait for popup
    MMI.waitForText("Low Tire Pressure", 2)
    except TimeOut:
      log(ERROR, "Low Tire Pressure popup timeout")
    ZBE.Press()
    # Callee ends call
    MobilePhoneControl.HangUp()
```

Listing 9.7. Executable test script.

9.5 Summary

We have seen in this chapter that task-based test case generation can be applied in practice. Furthermore, we have seen that *AspectT* is a powerful means for test case instantiation because it enables the combination of task sequence refinement and test script generation.

The state-machine-based refinement of a task model has an additional benefit: the generation of input sequences verifies whether the state machine behavior conforms to the task model. In our case study we found multiple errors in our dialog model where for a task sequence no corresponding test input sequence could be generated.

Test case generation is only one important part of model-based testing. The other important part is defining and implementing meaningful test oracles. At the time this thesis was created, the infotainment system was at the end-stage of its development and hence already well tested. Due to the lack of detailed test oracles, it was not possible to evaluate the generated test cases using the real system. Nevertheless, we consider the generation of inputs for test cases that cover feature interaction scenarios a first big step in the test case generation for infotainment systems. In the next chapter, we present an evaluation of our test selection criteria in order to show that the generated test cases cover critical scenarios.

10

Test Selection Criteria Evaluation

This chapter presents the last case study of task-based test case generation. The goal of this chapter is to show that test case generation from task models covers critical situations. We present the results of our evaluation of the introduced test selection criteria. The criteria are evaluated in the form of an empirical analysis of the faults that occurred during the development of a BMW infotainment system. The objective of this section is to show that the faults, which occurred in practice, would be detected with our task-based test case generation approach.

10.1 Goals

The goal of this case study is to evaluate our approach with respect to the faults that occurred during the development of a infotainment system at the BMW Group. More precisely, we want to answer two questions with this case study:

1. How many faults could be found with task-based test case generation?
2. How many faults could be found for each test selection criterion?

The first question aims at evaluating the concept of task-based feature modeling with respect to test case generation in general. The goal is to see if critical scenarios can be expressed by task models. The second question focuses on the proposed test selection criteria in Chapter 5. The goal is to see to which degree the proposed selection criteria cover critical situations.

10.2 Setup

The study is performed on the same data as the analysis in Chapter 3.2. The data is a collection of all faults that occurred during the development of a infotainment system a the BMW Group. The faults were taken from the bug-tracking system that was used during development. A fault is stored in the

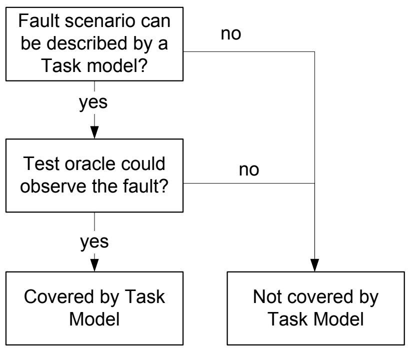

Fault scenario can be described by a Task model?
no
yes
Test oracle could observe the fault?
no
yes
Covered by Task Model
Not covered by Task Model

Fig. 10.1. Decision process.

bug-tracking system together with a description of its cause and its fix. These descriptions where analyzed in order to classify faults based on their cause. In this analysis we focus on the faults that were classified in the previous analysis (see Chapter 3.2) as caused by feature interactions.

In the first part of our study we classify faults into the two categories *covered by task model* and *not covered by task model*. The study is performed under the assumption that there exists a task model that describes all relevant features of the AIS. The first category contains all faults that occurred in a scenario that is described by a task model. The second category contains the faults that are not described by a task model. Furthermore, we assume that there are appropriate test oracles to observe the system's behavior. However, only faults were classified as detectable if the existence of a test oracle is reasonable. For example, we consider the assumption that there will be a test oracle that detects graphical errors during an animation as not reasonable. Figure 10.1 shows the decision process.

In the second part of our analysis, the faults are divided into different categories based on whether one of the introduced test selection criteria would generate a task sequence that covers the fault scenario. Hence, each test selection criterion represents a fault category and one category represents faults that would not not been covered by a test selection criterion.

10.3 Results and Discussion

Figure 10.2 shows the result of the first part of our analysis. The goal was to find out how many fault scenarios are detectable by task-based test case generation. The result shows that 88% of all feature interaction faults could have been detected by task-based test case generation and 12% could have been not.

An example of a fault that could not be found by task-based test case generation is the following: when the ignition is first set to on, then to off and then set back on before the bus transitions into the sleep mode, the audio functions were inactive even though they should be active. This fault could not be covered because it was caused by internal system behavior (the sleep mode of the

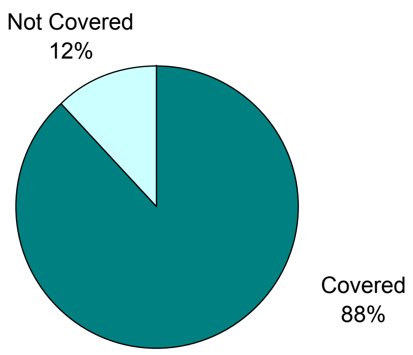


Fig. 10.2. Faults covered by task-based test case generation.

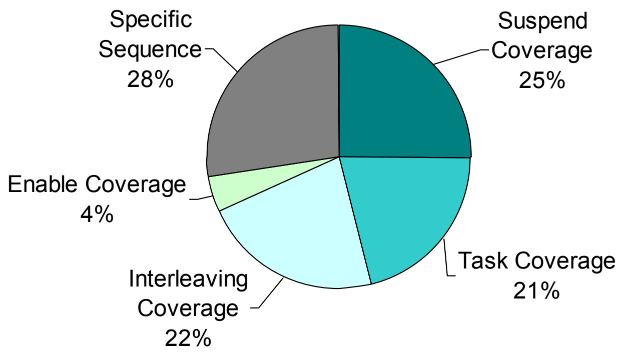


Fig. 10.3. Faults per test selection criterion.

bus) which would not be modeled by a task model. Another example of a fault which resulted from a feature interaction was the animation that did not run smoothly enough during the switch from the interactive map to the digital versatile disc (DVD) menu. This fault-scenario would have been modeled in a task model, however, the fault could not be found by an automated test oracle.

Nevertheless, our result clearly indicates that task models are an appropriate means for test case generation. Our previous analyses in chapter 3.2 showed that 40,9% of all severe faults result from feature interactions. In combination with our results from this case study we can conclude that task-based test case generation could cover 30% of all severe faults. This study reveals that task-based test case generation holds huge potential.

The second goal was to analyze, the number of faults found for each test selection criterion. The diagram in Figure 10.3 shows the results of the analysis. The diagram shows that *Suspend Coverage*, *Interleaving Coverage*, and *Task Coverage* find the most faults. 28% of the faults could not be found by the proposed test selection criteria and only 4% could be found by the *Enabling Coverage Criterion*.

The fact that the *task criterion* finds such a large fraction of faults is interesting, because it is a rather "trivial" test selection criterion. This results from the fact that a task sequence describes for each active task all modes of the other tasks. Many faults occurred in situations where a feature should have disabled another feature, but did not. An example of such a fault found by task coverage was that the GUI showed the CD selection dialog every time the voice guidance was active. This shows that unintentional feature interferences can be found by the task coverage criterion as well.

Another interesting result was the large number of faults found by the suspend coverage criterion. This was the case because there were many different interruption scenarios. Switching the ignition on and off while performing another task is especially error prone. An example of a fault resulting from an interruption between tasks was when the driver started the car while dialing a number. In some countries it is disallowed to dial a number while driving, therefore the system should have suspended the dialing until the car had stopped. The fault was that the driver could wrongfully dial a number while driving. This occurred when the driver started the car in the midst of dialing the number.

However, 28% of the faults could not be found by the proposed test selection criteria. These faults where classified as *specific scenarios*. These were scenarios in which a specific task sequence led to a fault which was not covered by one of the proposed test selection criteria. On the one hand, this indicates that there might be a potential for new test selection criteria that cover parts of these faults and on the other hand this advocates the application of random based test case generation.

Nevertheless, the study shows that more than two-thirds of all faults could be found by the proposed test selection criteria. 50% of the faults are covered by the straightforward test selection criteria *Suspend Coverage*, *Enable Coverage*, and *Task Coverage*. This is promising, because these three test selection criteria create relatively small test suites in comparison to the *Interleaving Coverage* criterion.

However, caveats need to be mentioned that can affect the validity of the result:

- The assumption that appropriate test oracles will be available might not be reasonable. Automated test case execution is gaining importance and hence future systems are designed to support automated testing by providing corresponding test interfaces. However, detecting graphical errors automatically is often not possible, especially in the context of animations. Hence, we classified graphical errors conservatively as non detectable.
- The sequences that lead to a fault are more complex than indicated by the bug descriptions. In order to avoid this situation we additionally included the descriptions of the corresponding bug fix in our evaluation. However, when we were not sure whether a task sequence was covered by a test selection criterion we classified the fault rather as not covered by a test selection criterion.

In this study, we focused only on faults that resulted from feature interactions. In addition, task-based test case generation could also be used to generate feature-specific test cases. We expect then the number of detected faults to be even higher.

10.4 Summary

The study shows that 88% of all error scenarios that resulted from feature interactions could be described by a task model. Furthermore, the study showed that the proposed test selection criteria cover a large fraction of faults. Together with the previous two case studies, this shows that task-based test case generation can be applied in practice and is able to systematically test feature interactions.

11

Summary and Conclusions

Infotainment systems are examples of multi-functional systems that integrate a large number of features. Over the last vehicle generations the number of features increased continuously. For example, an actual infotainment system in a BMW vehicle integrates up to 1000 different features. The number of features is expected to increase even more in future vehicle generations.

These features are realized in multiple electronic control units that are connected by communication busses. They share common resources, such as the user interface or communication channels. The shared usage of resources results in intentional and unintentional interactions between these features. In practice these interactions are often error prone. A study showed that 40% of all faults that occurred during the development of an infotainment system at the BMW Group resulted from erroneous feature interactions. In order to cope with feature interactions and to detect erroneous interactions early, the interactions between features must be tested thoroughly. However, up to now there are no approaches that enable a systematic and automated test of feature interactions.

In this thesis we targeted this situation and presented an approach for test case generation that enables the systematic coverage of feature interaction scenarios in order to cope with the complexity of future infotainment systems.

11.1 Summary

Model-based test case generation involves multiple steps. First of all, an appropriate *test model* is required that describes critical system properties and that omits unnecessary detail. We introduce task models as means to model features and their interactions. Task models provide a simple graphical notation for the explicit modeling of feature interactions. Such a task model describes the space of all possible feature interaction scenarios.

The second step in model-based test case generation is to generate appropriate test cases. We propose multiple *test selection criteria* that derive task sequences from a task model. These criteria select task sequences that systematically cover intentional and unintentional feature interaction scenarios.

Task models describe the interaction between environment and system on a high abstraction level. This reduces the state space of large systems to a feasible size. Furthermore, they focus on critical feature interaction scenarios by omitting unnecessary details. However, the generated task sequences lack two elementary parts of a test case: they describe no input and no output behavior. Hence, the third step of model-based test case generation, the *test case instantiation*, is one of the most important steps. Test cases are instantiated into executable test scripts in order to execute them automatically. This enables a continuous testing process during development. Furthermore, this is an important prerequisite in order to cope with a potentially large number of generated test cases.

As a solution to this problem, we propose two approaches for test case instantiation: *task sequence refinement* and *aspect-oriented test script generation*. The first approach is specific to task models. Our approach of task sequence refinement transforms task sequences into test cases by enriching them with additional input and output behavior. The missing information is derived from behavior models on a lower abstraction level.

The second approach can be applied to generate test scripts from abstract test cases that are defined in arbitrary modeling languages. The problem is that test script generation varies depending on different testing concerns, such as test goal, test setup and test phase. Thus, for each testing concern a new transformation must be defined. We propose aspect-orientation as means to modularize these testing concerns. The modularization of testing concerns reduces test case instantiation effort by enabling their reuse in different testing contexts.

To conclude, in this thesis we propose an integrated approach for test case generation that involves all steps to systematically test feature interactions. We showed the feasibility and suitability of our approach by multiple case studies in the automotive domain. Furthermore, we created an integrated tooling environment that seamlessly combines all approaches to enable their application in practice.

11.2 Conclusion

The process of task-based test case generation has been created as a part of the preparation for future generations of infotainment systems. All case studies were taken from the automotive domain in order to demonstrate the feasibility for real world applications. However, task-based test case generation is not restricted to the automotive domain. The approach can be applied to any multi-functional system that is characterized by a strong interaction with its environment.

We consider the explicit modeling of feature interaction to be an important step to handle the complexity of future infotainment systems. The first advantage is that solely by the process of modeling feature interactions, prior unspecified feature interactions are likely to be identified. Furthermore, using task models enables an early specification of features and their interactions.

Model-based test case generation is inevitably linked with test case instantiation. Considerable effort is required to set up an environment for automated test case execution. This is especially true for embedded systems. However, one of the most important aspects is to design the system for testability in order to support automated testing by providing appropriate test interfaces. We introduced *AspectT* in the context of task-based test case generation and it has proven as an appropriate language for test case instantiation in our case study. One of the key features of *AspectT* is its flexibility to support different modeling languages. Hence, it is applicable in any context where abstract test cases must be translated into test scripts.

In this thesis we created an approach for test case generation from task models. The approach involved all necessary steps from modeling the system, selecting test cases, and generating executable test scripts. However, there are still open problems that require further research.

11.3 Future Work

We were able to model large portions of an actual infotainment system using *TTask*. However, we also identified certain scenarios that could not be modeled in *TTask*, for example, when the same task can be performed multiple times in parallel. One task for the future work is to identify these situations and to extend *TTask* accordingly.

We presented a case study that generated executable test cases for testing an existing system. However, due to missing test oracles, the actual test of the system was not possible. A goal for future work is to perform a case study, in which a real system is tested using our approach in order to determine the real fraction of feature interaction faults that can be detected. Furthermore, it would be interesting to apply our approach in another domain in order to show that it is not specific to the automotive domain.

In this thesis we focused on designing and realizing our approach of task-based test case generation. However, finding the best way of integrating task-based test case generation into the current development process at BMW is still an open issue.

A

Appendix

A.1 *TTask* to Promela transformation

A.1.1 *TTask* EMF Metamodel

The *TTask* metamodel is defined using Ecore which is part of the Eclipse Modeling Framework (EMF)[1]. Figure A.1 shows a class diagram of the *TTask* metamodel. EMF is used in combination with the Graphical Modeling Framework (GMF)[2] to implement the graphical editor of the *TTask* modeling environment. Furthermore, the *TTask* to Promela transformer is implemented based on the Ecore metamodel.

A.1.2 Promela Code Generator

The *TTask* to Promela transformation is performed by generating Promela code from a given *TTask* model. The code generator is implemented using XPAND which is part of the Openarchitectureware framework[3]. Listing A.1 shows the implementation of the code generator.

```
«IMPORT task»
«EXTENSION extend::TaskPromela»

«DEFINE main FOR Collection[Task]»
«FILE "pan_in.prm"»
«EXPAND init FOR this»
«EXPAND user FOR this»

«ENDFILE»
«ENDDEFINE»

«DEFINE init FOR Collection[Task]»
mtype = {
```

[1] http://www.eclipse.org/emf/
[2] http://www.eclipse.org/gmf/
[3] http://openarchitectureware.org/

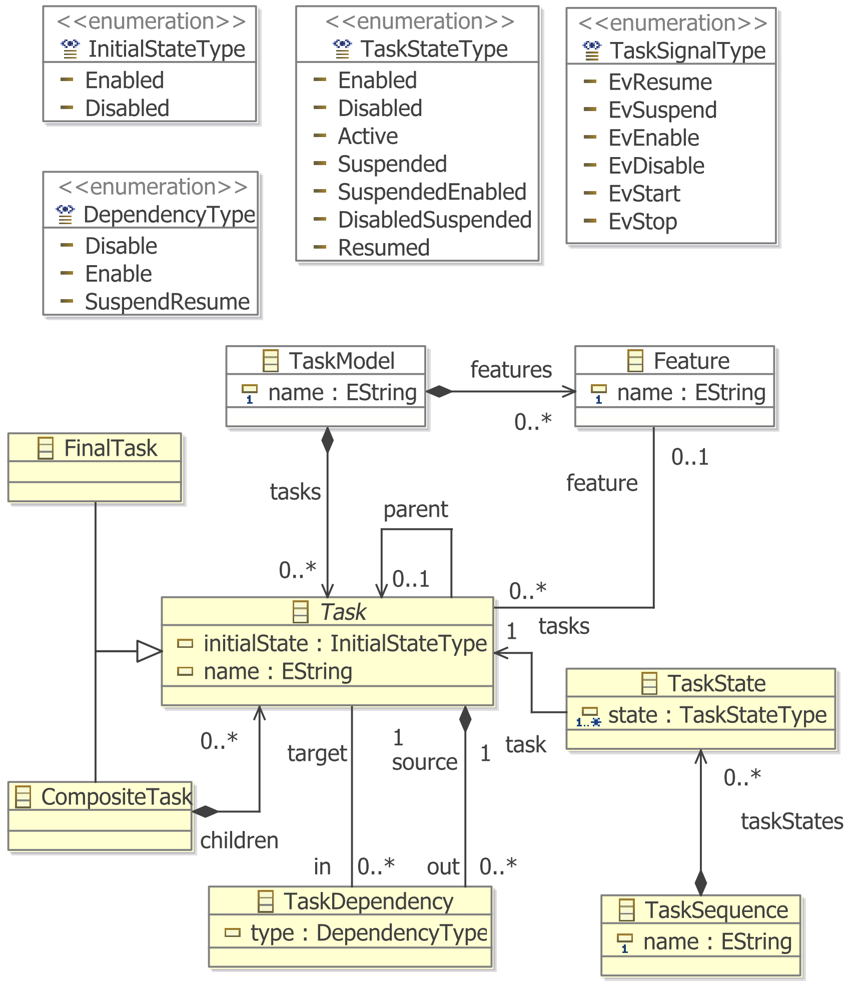


Fig. A.1. TTask EMF Metamodel.

```
«FOREACH getTaskStateTypes() AS d SEPARATOR ","»«d.toString()
    »«ENDFOREACH»
«FOREACH getTaskSignalTypes() AS d SEPARATOR ","»«d.toString()
    »«ENDFOREACH»
 };

«FOREACH this AS task»
chan in«task.getName()»  = [0] of {mtype};
«ENDFOREACH-»
chan inEnvironment   = [0] of {bool};
«FOREACH this.select(e|e.hasMultipleSuspendDependencies()) AS task-»
int «task.getName()»Suspend = 0;
«ENDFOREACH-»

«FOREACH this AS task-»
«IF task.isInitiallyEnabled()-»
mtype «task.getName()»State = «InitialStateType::Enabled»;
«ELSE-»
```

```
mtype «task.getName()»State = «InitialStateType::Disabled»;
«ENDIF-»

«ENDFOREACH-»

«EXPAND task FOREACH this»
«ENDDEFINE»

«DEFINE user FOR Collection[Task]»
active proctype Environment()
{
   do
       «FOREACH this.select(i | i.isLeaf()) AS task-»
           :: «task.getName()»State == «TaskStateType::Enabled» ->
               «EXPAND send(TaskSignalType::EvStart, "Environment") FOR
                    task»
               timeout;
               /*inEnvironment?ready; */
               «EXPAND send(TaskSignalType::EvStop, "Environment") FOR
                   task»
               /*inEnvironment?ready; */
               timeout;
       «ENDFOREACH-»
          :: else -> skip;

   od;
}

«ENDDEFINE»



«DEFINE task FOR Task»
proctype «getName()»()
{
  atomic{
  mtype new;
  in«getName()»?new ->
  if
    «IF isLeaf()»
    :: new == «TaskSignalType::EvStart» && «getName()»State ==
        «TaskStateType::Enabled» ->
      «EXPAND start FOR this-»

    :: new == «TaskSignalType::EvStop» && «getName()»State ==
        «TaskStateType::Active» ->
      «EXPAND stop FOR this-»
    «ELSE»
    :: new == «TaskSignalType::EvStart» && «getName()»State !=
        «TaskStateType::Active» ->
      «EXPAND start FOR this-»

    :: new == «TaskSignalType::EvStop» «FOREACH children AS c» && «c.
        getName()»State != «TaskStateType::Enabled» && «c.getName()
        »State != «TaskStateType::Active» «ENDFOREACH»->
      «EXPAND stop FOR this-»
   «ENDIF»

   :: new == «TaskSignalType::EvDisable» ->
     «EXPAND disable FOR this-»
   :: new == «TaskSignalType::EvEnable» ->
     «EXPAND enable FOR this-»
   :: new == «TaskSignalType::EvResume» ->
     «EXPAND resume FOR this-»
   :: new == «TaskSignalType::EvSuspend» ->
     «EXPAND suspend FOR this-»
   :: else -> skip;
  fi;
  }
}
«ENDDEFINE»

«DEFINE start FOR Task»

  «IF parent != null-»
  if
   :: «parent.getName()»State != «TaskStateType::Active» ->
     «EXPAND send(TaskSignalType::EvStart, getName()) FOR parent»
   :: else -> skip;
  fi;
  «ENDIF-»
  «REM»«this.getName()»Coverage = «TaskStateType::Active»;«ENDREM»
  «EXPAND changeState(TaskStateType::Active, TaskStateType::Active) FOR
       this»
  «EXPAND send(TaskSignalType::EvDisable, getName()) FOREACH
      getDisabledTasks()»
   «EXPAND send(TaskSignalType::EvSuspend, getName()) FOREACH
       getSuspendedTasks()»
«ENDDEFINE»

«DEFINE stop FOR Task»
  «EXPAND changeState(TaskStateType::Disabled, TaskStateType::Stopped)
      FOR this»

  «FOREACH getEnabledTasks() AS enabledTask»
  «EXPAND send(TaskSignalType::EvEnable, getName()) FOR enabledTask»
  «ENDFOREACH»

  «EXPAND send(TaskSignalType::EvResume, getName()) FOREACH
      getSuspendedTasks()»
  «IF parent != null-»
   «EXPAND send(TaskSignalType::EvStop, getName()) FOR parent»
  «ENDIF-»


«ENDDEFINE»


«DEFINE enable FOR Task»
  if
   :: «getName()»State == «TaskStateType::Disabled» ->
```

```
      «EXPAND changeState(TaskStateType::Enabled, TaskStateType::
          Enabled) FOR this»
     :: «getName()»State == «TaskStateType::DisabledSuspended» ->
      «EXPAND changeState(TaskStateType::SuspendedEnabled,
          TaskStateType::SuspendedEnabled) FOR this»
    :: else -> skip;
  fi;
  «EXPAND send(TaskSignalType::EvEnable, getName()) FOREACH
      getInitialChildTasks()»
«ENDDEFINE»

«DEFINE disable FOR Task»
  «EXPAND send(TaskSignalType::EvDisable, getName()) FOREACH children»
  if
    :: «getName()»State == «TaskStateType::Enabled» ->
      «EXPAND changeState(TaskStateType::Disabled, TaskStateType::
          Disabled) FOR this»
    :: «getName()»State == «TaskStateType::Suspended» ->
      «EXPAND changeState(TaskStateType::DisabledSuspended,
          TaskStateType::DisabledSuspended) FOR this»
      «EXPAND send(TaskSignalType::EvResume, getName()) FOREACH
          getSuspendedTasks()»
    :: «getName()»State == «TaskStateType::SuspendedEnabled» ->
      «EXPAND changeState(TaskStateType::DisabledSuspended,
          TaskStateType::DisabledSuspended) FOR this»
    :: else -> skip;
  fi;

«ENDDEFINE»

«DEFINE suspend FOR Task»
  if
    :: «getName()»State == «TaskStateType::Enabled» ->
      «EXPAND changeState(TaskStateType::SuspendedEnabled,
          TaskStateType::SuspendedEnabled) FOR this»
    :: «getName()»State == «TaskStateType::Active» ->
      «EXPAND changeState(TaskStateType::Suspended, TaskStateType::
          Suspended) FOR this»
    :: «getName()»State == «TaskStateType::Disabled» ->
      «EXPAND changeState(TaskStateType::DisabledSuspended,
          TaskStateType::DisabledSuspended) FOR this»
     «IF hasMultipleSuspendDependencies()»
    ::  «getName()»State == «TaskStateType::SuspendedEnabled»
      || «getName()»State == «TaskStateType::Suspended»
      || «getName()»State == «TaskStateType::DisabledSuspended»
      «getName()»Suspend = «getName()»Suspend + 1;
    «ENDIF»
    :: else -> skip;
  fi;
  «EXPAND send(TaskSignalType::EvSuspend, getName()) FOREACH children»
«ENDDEFINE»

«DEFINE resume FOR Task»
«IF hasMultipleSuspendDependencies()-»
  printf("Status: «getName()»Suspend: %d \n", «getName()»Suspend);
  if
```

```
    :: «getName()»Suspend == 0 ->
«ENDIF-»
    if
      :: «getName()»State == «TaskStateType::Suspended» ->
        «EXPAND changeState(TaskStateType::Active, TaskStateType::
            Resumed) FOR this»
      :: «getName()»State == «TaskStateType::SuspendedEnabled» ->
        «EXPAND changeState(TaskStateType::Enabled, TaskStateType::
            Enabled) FOR this»
      :: «getName()»State == «TaskStateType::DisabledSuspended» ->
        «EXPAND changeState(TaskStateType::Disabled, TaskStateType::
            Disabled) FOR this»
      :: else -> skip;
    fi;
«IF hasMultipleSuspendDependencies()-»
    :: else -> «getName()»Suspend = «getName()»Suspend - 1;
  fi;
«ENDIF-»
  «EXPAND send(TaskSignalType::EvResume, getName()) FOREACH children»
«ENDDEFINE»

«DEFINE send(TaskSignalType signal, String source) FOR Task»
printf("SEND: «source»-«signal»-«this.getName()»\n");
run «this.getName()»();
in«this.getName()»!«signal»;
«ENDDEFINE»

«DEFINE changeState(TaskStateType state, TaskStateType print) FOR Task»
      printf("MSC: «getName()»=«print»\n");
      «getName()»State = «state»;
«ENDDEFINE»

«DEFINE activeCoverageNeverClaim FOR Collection[Task]»
«FOREACH this AS task-»
mtype «task.getName()»Coverage = «InitialStateType::Disabled»;
«ENDFOREACH-»

never {
do
    ::
    «FOREACH this AS task SEPARATOR "&&"»
       «task.getName()»Coverage = Active
    «ENDFOREACH»-> break
    :: else
od
}
«ENDDEFINE»
```

Listing A.1. Promela code generator.

References

[Abr96] J. R. Abrial. *The B-Book. Assigning programs to meaning.* Cambridge University Press, 1996.

[ACG+00] D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien, and T. Ware. Feature description and feature interaction analysis with use case maps and lotos, 2000.

[ALW05] Daniel Amyot, Luigi Logrippo, and Michael Weiss. Generation of test purposes from use case maps. *Comput. Netw.*, 49(5):643–660, 2005.

[Amy03] Daniel Amyot. Introduction to the user requirements notation: learning by example. *Comput. Netw.*, 42(3):285–301, 2003.

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

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[BBM03] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework.* Pearson Education, 2003.

[BC96] R. J. A. Buhr and R. S. Casselman. *Use case maps for object-oriented systems.* Prentice Hall, 1996.

[BDD+92] Manfred Broy, Frank Dederich, Claus Dendorfer, Max Fuchs, Thomas Gritzner, and Rainer Weber. The Design of Distributed Systems - An Introduction to FOCUS. Technical Report TUM-I9202, Technische Universität München, jan 1992.

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

[Ben08] Sebastian Benz. AspectT: Aspect-oriented test case instantiation. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 1–12, New York, NY, USA, 2008. ACM.

[BFdV+99] Axel Belinfante, Jan Feenstra, René G. de Vries, Jan Tretmans, Nicolae Goga, Loe M. G. Feijs, Sjouke Mauw, and Lex Heerink. Formal test automation: A simple experiment. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems*, pages 179–196. Kluwer, 1999.

[BGK00] L. G. Bouma, Nancy Griffeth, and Kristofer Kimbler. Feature interactions in telecommunications systems. *Comput. Netw.*, 32(4):383–387, 2000.

[BK98] Manfred Broy and Ingolf Krüger. Interaction interfaces - towards a scientific foundation of a methodological usage of message sequence charts. In *ICFEM '98: Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):5, 2007.

[BL03] Fabrice Bouquet and Bruno Legeard. Reification of executable test scripts in formal specification-based test generation: The java card transaction mechanism case study. In *Proc. of FME'03, Formal Method Europe*, volume 2805 of *LNCS*, pages 778–795, Pisa, Italy, September 2003.

[BOS91] BOSCH. *CAN Specification Version 2.0.* Robert BOSCH GmbH, 1991.

[BR04] Purandar Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions, 2004.

[BS01] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer, 2001.

[Buh98] R. J. A. Buhr. Use case maps as architectural entities for complex systems. *IEEE Trans. Softw. Eng.*, 24(12):1131–1155, 1998.

[CCI84] CCITT. Functional specification and description language (SDL). Recommendations Z. 100 - Z. 104, 1984.

[CGL$^+$94] Jane Cameron, Nancy D. Griffeth, Yow-Jian Lin, Margaret E. Nilson, William K. Schnure, and Hugo Velthuijsen. A feature interaction benchmark for in and beyond. In Wiet Bouma and Hugo Velthuijsen, editors, *FIW*, pages 1–23. IOS Press, 1994.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

[CH07] Caixia Chi and Ruibing Hao. Test generation for interaction detection in feature-rich communication systems. *Comput. Netw.*, 51(2):426–438, 2007.

[Cho07] Yunja Choi. From NuSMV to SPIN: Experiences with model checking flight guidance systems. *Form. Methods Syst. Des.*, 30(3):199–216, 2007.

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Netw.*, 41(1):115–141, 2003.

[CLOM07] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 84–94, New York, NY, USA, 2007. ACM.

[DGP$^+$04] Martin Deubler, Johannes Grünbauer, Gerhard Popp, Guido Wimmel, and Christian Salzmann. Towards a Model-Based and Incremental Development Process for Service-Based Systems. In *Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2004), Innsbruck*, 2004.

[DN84] W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, vol. SE-10:pp. 438–444, July 1984.

[DS03] Dan Diaper and Neville Stanton. *The Handbook of Task Analysis for Human-Computer Interaction.* Lawrence Erlbaum Associates, 2003.

[EKRV06] Juhan P. Ernits, Andres Kull, Kullo Raiend, and Jüri Vain. Generating tests from EFSM models using guided model checking and iterated search refinement. pages 85–99, 2006.

[EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Trans. Softw. Eng.*, 29(3):210–224, 2003.

[FAM06] André L. L. Figueiredo, Wilkerson L. Andrade, and Patrícia D. L. Machado. Generating interaction test cases for mobile phone systems from use case specifications. *SIGSOFT Softw. Eng. Notes*, 31(6):1–10, 2006.

[FL00] Peter Fröhlich and Johannes Link. Automated test case generation from dynamic models. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 472–492, London, UK, 2000. Springer-Verlag.

[FN03] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automated conflict detection. *ACM Trans. Softw. Eng. Methodol.*, 12(1):3–27, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[GHR+03a] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (ttcn-3). *Comput. Netw.*, 42(3):375–403, 2003.

[GHR+03b] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction into the testing and test control notation (TTCN-3). *Computer Networks, Volume 42, Issue 3.*, pages 375–403, June 2003.

[GL93] Nancy D. Griffeth and Yow-Jian Lin. Extending telecommunications systems: The feature-interaction problem. *Computer*, 26(8):14–18, 1993.

[GLZ06] Jeff Gray, Yuehua Lin, and Jing Zhang. Automating change evolution in model-driven engineering. *Computer*, 39(2):51, 2006.

[HCL+03] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, and Hasan Ural. Data flow testing as model checking. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 232–242, Washington, DC, USA, 2003. IEEE Computer Society.

[HdMR05] Gregoire Hamon, Leonardo de Moura, and John Rushby. Automated test generation with SAL. Technical report, Computer Science Laboratory, 2005.

[HGW04] Mats P.E. Heimdahl, Devaraj George, and Robert Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? *High Assurance Systems Engineering*, 00:178–186, 2004.

[HIM00] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. UML-Based integration testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 60–70, New York, NY, USA, 2000. ACM Press.

[HKU04] Rob M. Hierons, T.-H. Kim, and Hasan Ural. On the testability of SDL specifications. *Comput. Netw.*, 44(5):681–700, 2004.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[HR04] Gregoire Hamon and John Rushby. An operational semantics for stateflow. *Int. J. Softw. Tools Technol. Transf.*, 9(5):447–456, 2004.

[HRV+03] Mats P. E. Heimdahl, Sanjai Rayadurgam, Willem Visser, Devaraj George, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In *Third International International Workshop on Formal Approaches to Software Testing*, volume Lecture Notes in Computer Science. Springer, 2003.

[HSH90] H. Rex Hartson, Antonio C. Siochi, and D. Hix. The UAN: a user-oriented representation for direct manipulation interface designs. *ACM Trans. Inf. Syst.*, 8(3):181–203, 1990.

[IEE04] IEEE. Software engineering body of knowledge (swebok), 2004.

[ITU96] ITU. Message sequence charts. ITU-T recommendation Z.120, 1996.

[Jal91] Pankaj Jalote. *An integrated approach to software engineering.* Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[JJ05] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005.

[JK96a] Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3(4):320–351, 1996.

[JK96b] Bonnie E. John and David E. Kieras. Using GOMS for user interface design and evaluation: which technique? *ACM Trans. Comput.-Hum. Interact.*, 3(4):287–319, 1996.

[KA92] B. Kirwan and L.K. Ainsworth. *A guide to task analysis.* Taylor and Francis, London, 1992.

[KCK94] B. Kelly, M. Crowther, and J. King. Feature interaction detection using SDL models. *Global Telecommunications Conference, 1994. GLOBECOM '94. Communications: The Global Bridge., IEEE*, 3:1857–1861 vol.3, Nov- 2 Dec 1994.

[Kic96] Gregor Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[KK98] Dirk O. Keck and Paul J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, 1998.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[LP04] Vitus S. W. Lam and Julian Padget. Formal specification and verification of the set/a protocol with an integrated approach. In *CEC '04: Proceedings of the IEEE International Conference on E-Commerce Technology*, pages 229–235, Washington, DC, USA, 2004. IEEE Computer Society.

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

[Met04] Andreas Metzger. Feature interactions in embedded control systems. *Comput. Netw.*, 45(5):625–644, 2004.

[MJW00] Oege Moor, Simon L. Peyton Jones, and Eric Van Wyk. Aspect-oriented compilers. In *GCSE '99: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, pages 121–133, London, UK, 2000. Springer-Verlag.

[MOS] MOST Cooperation. *MOST Specification.*

[MPS02] Giulio Mori, Fabio Paternò, and Carmen Santoro. CTTE: Support for developing and analyzing task models for interactive system design. *IEEE Transactions on Software Engineering*, 28(8):797–813, 2002.

[MW03] Andreas Metzger and Christian Webel. Feature interaction detection in building control systems by means of a formal product model. In *Proc. of Feature Interaction in Telecommunications and Software Systems VII*, pages 105–121. Press, 2003.

[MXX06] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for UML activity diagrams. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 2–8, New York, NY, USA, 2006. ACM.

[NNC05] Leonel Nóbrega, Nuno Jardim Nunes, and Helder Coelho. Mapping ConcurTaskTrees into UML 2.0. In Stephen W. Gilroy and Michael D. Harrison, editors, *DSV-IS*, volume 3941 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 2005.

[Obj02] Object Management Group (OMG). Meta object facility (MOF) specification. formal/2002-04-03, April 2002.

[OMG03] Object Management Group. *OMG Unified Modeling Language Specification*, März 2003.

[OXL99] A Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *ICECCS '99: Proceedings of the 5th International Conference on Engineering of Complex Computer Systems*, page 119, Washington, DC, USA, 1999. IEEE Computer Society.

[Paa95] Jukka Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.

[Pat99] Fabio Paternò. *Model-Based Design and Evaluation of Interactive Applications.* Springer-Verlag, London, UK, 1999.

[PERH04] Wolfgang Prenninger, Mohammad El-Ramly, and Marc Horstmann. Case studies. In *Model-Based Testing of Reactive Systems*, pages 439–461, 2004.

[PFH$^+$06] Christian Pfaller, Andreas Fleischmann, Judith Hartmann, Martin Rappl, Sabine Rittmann, and Doris Wild. On the Integration of Design and Test - A Model Based Approach for Embedded Systems. *Proceedings of the Workshop on Automation of Software Test (AST 06)*, 2006.

[PMM97] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. ConcurTaskTrees: A diagrammatic notation for specifying task models. In *INTERACT*, pages 362–369, 1997.

[PP05] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. *Electr. Notes Theor. Comput. Sci.*, 116:59–71, 2005.

[RH01] Sanjai Rayadurgam and Mats P. E. Heimdahl. Coverage based test-case generation using model checkers. *Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS '01)*, 0:0083, 2001.

[Rit08] Sabine Rittmann. *A methodology for modeling usage behavior of multi-functional systems.* PhD thesis, Technische Universität München, 2008.

[Sti99] R. E. Kurt Stirewalt. MDL: A language for binding user-interface models. In *CADUI*, pages 159–170, 1999.

[SY98] N. A. Stanton and M. Young. Is utility in the eye of the beholder? a study of ergonomics methods. *Applied Ergonomics*, 29(1):21–54, 1998.

[TB96] Jean-Claude Tarby and Marie-France Barthet. The DIANE+ method. In Jean Vanderdonckt, editor, *CADUI*, pages 95–120. Presses Universitaires de Namur, 1996.

[TSL04] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. *Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on*, pages 493–498, Nov. 2004.

[UL06] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[Vel93] Hugo Velthuijsen. Distributed artificial intelligence for runtime feature-interaction resolution. *Computer*, 26(8):48–55, 1993.

[Vis01] E. Visser. Stratego: A language for program transformation based on rewriting strategies (system description). In A. Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference*, LNCS 2051. Springer, 2001.

[WE04] Michael Weiss and Babak Esfandiari. On feature interactions among web services. *ICWS*, page 88, 2004.

[WEL07] M. Weiss, B. Esfandiari, and Y. Luo. Towards a classification of web service feature interactions. *Comput. Netw.*, 51(2):359–381, 2007.

[Wol08] Elisabeth Wolf. Specification-based test case generation for an automotive HMI. Master's thesis, TU München, 2008.

[Zav01] Pamela Zave. Feature-oriented description, formal methods, and DFC. In *Language Constructs for Describing Features*, pages 11–26. Springer-Verlag London Ltd., 2001.

[Zav03] Pamela Zave. An experiment in feature engineering. *Programming methodology*, pages 353–377, 2003.

[ZML07] Hongwei Zeng, Huaikou Miao, and Jing Liu. Specification-based test generation and optimization using model checking. In *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 349–355, Washington, DC, USA, 2007. IEEE Computer Society.

[ZZK07] Yongyan Zheng, Jiong Zhou, and Paul Krause. An automatic test case generation framework for web services. *JSW*, 2(3):64–77, 2007.