

# AspectT: Aspect-Oriented Test Case Instantiation

Sebastian Benz

BMW Car IT GmbH

Sebastian.Benz@bmw-carit.de

## Abstract

Test case instantiation is the transformation of abstract test cases into executable test scripts. Abstract test cases are either created during model based test case generation or are manually defined in a suitable modeling notation. The transformation 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. This paper introduces AspectT, an aspect-oriented language for the instantiation of abstract test cases. We reduce the effort of test case instantiation by modularizing testing concerns in the form of aspects to enable their reuse in different testing contexts. The approach is implemented and integrated in an existing testing framework and has been successfully applied to test an electronic control unit of an automotive infotainment system at BMW Group.

**Categories and Subject Descriptors** D.2.4 [Software/Program Verification]: Validation; D.2.5 [Testing and Debugging]: Testing tools; D.3.3 [Programming Languages]: Language Constructs and Features

**General Terms** Model-Based Testing, Test Case Generation, Test Case Instantiation, Aspect-Oriented

## 1. Introduction

In software development practice, model-based testing is gaining influence. There are two ways of using models for testing: model-based test case specification and model-based test case generation. Model-based test case specification relies on modeling languages such as Message Sequence Charts MSC [1], to describe abstract test cases. Model-based test case generation automates the creation of test cases. An environment or behavior model is used to generate test cases. Another application of model-based testing is to generate test input data from a domain model, where test generation involves the selection and combination of different test inputs. In this paper, we concentrate on the problem of making abstract test cases executable. This problem needs to be solved for both model-based test case specification and generation.

Both model-based test case specification and model-based test case generation result in abstract test cases. It is desirable to automate the execution of these test cases to reduce manual test execution effort and to enable a continuous testing process. The automated execution requires additional information to bridge the ab-

straction gap between test case and the System Under Test (SUT). 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. An example of an observable system property is that the graphical user interface (GUI) signals an incoming call on the screen. 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* [16] 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 driver components, which encapsulate the missing information (e.g., a wrapper for user inputs). In practice, mixed approaches are common where test cases are translated into test scripts that trigger certain driver components.

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. In the case studies in [17], test case instantiation took about 25-45% of the modeling time. 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD’08, March 31 – April 4, 2008, Brussels, Belgium.  
Copyright © 2008 ACM 978-1-60558-044-9/08/0003...\$5.00

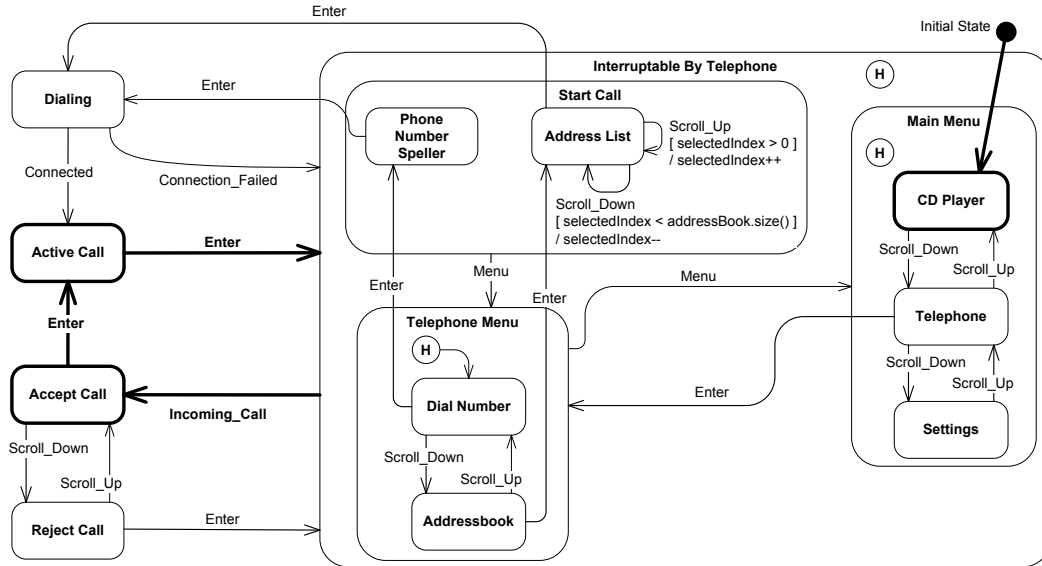


Figure 1. GUI statechart model.

The approach presented in this paper applies principles of aspect-orientation to the problem of test case instantiation in order to reduce the effort of test case instantiation. Modularizing test oracle definition, input mapping and configuration into separate aspects enables their reuse in different scenarios that focus on different testing concerns. The contributions of this paper 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.

The paper 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 for aspect-oriented test case instantiation, 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.

## 2. Test Case Instantiation

In this section we introduce test case instantiation. We start with an example from the automotive domain where we used UML statecharts<sup>1</sup> to generate test cases. Figure 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 6.

Each path in the model is a potential test case. The bold transitions in Figure 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 joystick (Enter) the call is accepted (Active Call). To finish the call, the user presses the joystick 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 2. The test 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.

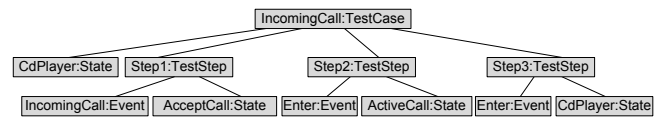


Figure 2. Test case for the “Incoming Call” use case.

Currently there are two main approaches to test case instantiation [17]: 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.

<sup>1</sup> <http://www.omg.org/uml/>

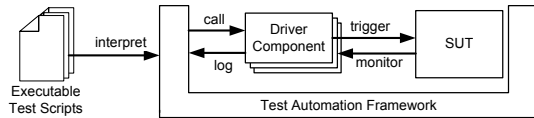


Figure 3. Test automation framework and driver components.

In practice, mixed approaches are common where driver components raise the SUT’s abstraction level and the test cases are translated into test scripts that use these driver components during execution (see Figure 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.<sup>2</sup> These scripts use the driver components to stimulate and observe the system. The event “Enter” in the test case in Figure 2 is translated into the corresponding test script code that calls the joystick 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 1 shows such a test oracle definition in the form of a test script snippet. 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 1. Test oracle for “CD Player” state.

One test case can be used to test different system properties. A test generated from the example in Figure 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 4 shows an example of dependencies between test case, test scripts and testing concerns. For a given test case, four test scripts are 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 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 paper 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 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,

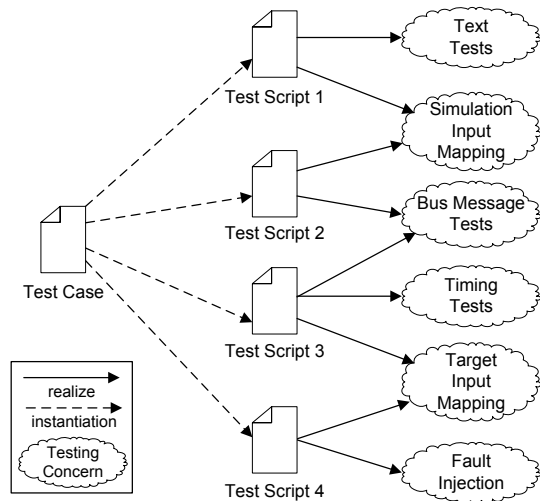


Figure 4. One test case - many test scripts.

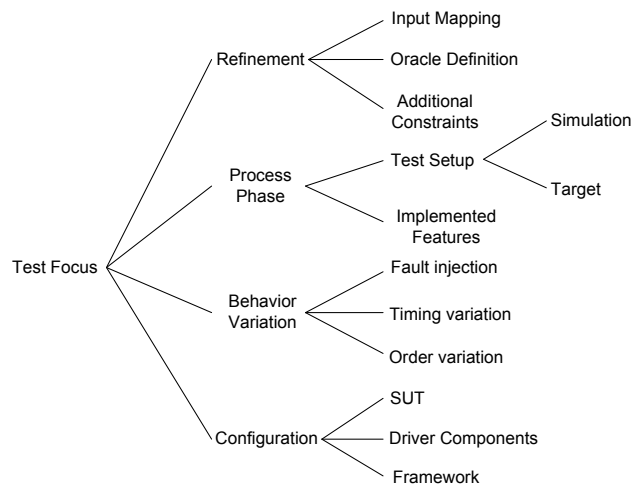


Figure 5. Classification of test instantiation concerns.

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 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 sim-

<sup>2</sup> <http://www.python.org>

ulated or real. With increasing development progress the system properties that must be tested become more complex, whereas 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*.

The factors described in the previous paragraphs 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.

### 3. Approach

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 1 and 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. Furthermore, in the last section we introduced 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.

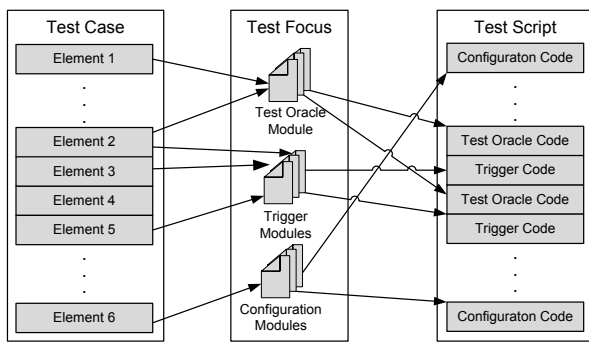


Figure 6. Transformation of a test case into a test script.

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 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 6 shows these relations between test focus modules, test case and test script.

Aspect-oriented software development approaches [12] 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.

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 2). Each node in the test case tree is a potential join point where aspects emit chunks of code into the test script. The resulting test script is the composition of all code chunks that were emitted during weaving.

Similarly to common 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 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 implement additional constraints or triggers are invoked either *before*, *after* or *around* a join point.

In the previous section, Listing 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 node that can occur in a test case tree, such as State or Event in Figure 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.

## 4. 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.

### 4.1 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 languages (DSL) are gaining influence in development, specification and hence in model-based test case generation. The case study presented in this paper 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) [14] 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 [6] language which is part of the Eclipse Modeling Framework (EMF). Ecore implements a subset of MOF and is tightly integrated into the Eclipse IDE.<sup>3</sup> 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.<sup>4</sup> 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 2. Figure 7 shows the test case metamodel 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

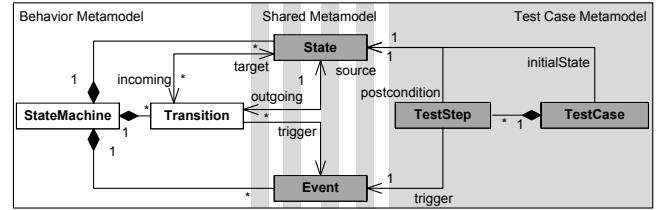


Figure 7. Test case metamodel.

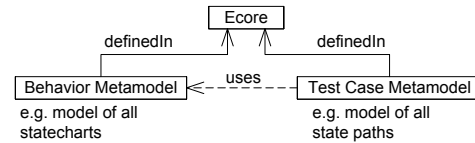


Figure 8. Behavior and test case model are defined in Ecore.

case sequence is defined by a depth-first left-right tree walk. A test case is either described by a dedicated test case 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, 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 6. Figure 8 shows the relationship between Ecore, behavior model and test case model.

By using Ecore models as join point models, it is possible to use AspectT with different test case models. 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 [10]. The downside of using a modeling 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.

### 4.2 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., TTCN3 [8]). 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 wo-

<sup>3</sup> <http://www.eclipse.org>

<sup>4</sup> <http://www.eclipse.org/modeling/mdt/?project=uml2>

ven 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 2 shows an example aspect definition. The weaving happens during a tree walk through the test case. The aspect writes “print ‘Hello World!’” for the first time it is woven. Any further time it is woven, it writes the state’s name and “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 = <% true %>
  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 2. Sample aspect definition.

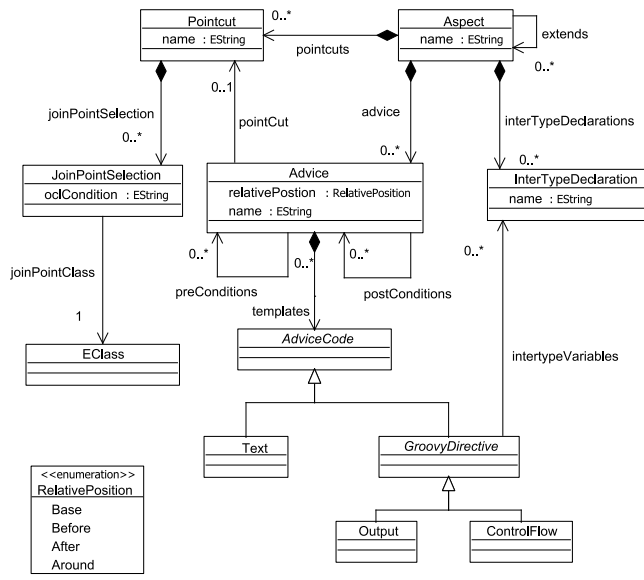


Figure 9. The Ecore metamodel of AspectT.

The AspectT tool has also been implemented using EMF. Figure 9 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.

#### 4.2.1 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 10 shows an example for a join point selection. A pointcut can contain multiple join point selection statements. The constraint is defined using the Object Con-

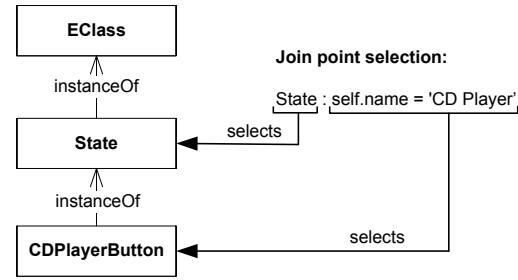


Figure 10. Pointcut definition in AspectT.

straint Language (OCL),<sup>5</sup> where *self* refers to the current join point object. OCL was chosen due to its powerful query expressions.

#### 4.2.2 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<sup>6</sup>. 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<sup>7</sup> or PHP.<sup>8</sup> An advice defines text that is emitted into the 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 2 we used directives to write the text depending on the 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.
- 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

<sup>5</sup> <http://www.omg.org/docs/ptc/03-10-14.pdf>

<sup>6</sup> <http://groovy.codehaus.org/>

<sup>7</sup> <http://java.sun.com/products/jsp/>

<sup>8</sup> <http://www.php.net/>

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.

### 4.2.3 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. Superspects are defined using the keyword *extends*. In this case all superspects' advice are woven as well. Additionally an aspect can define its own advice for pointcuts defined in the superspect. It is possible to define precedence relations between advice in the subspect and in the superspect. Aspect inheritance allows the reuse of pointcuts and advice across different aspects. Figure 11 shows an example of two aspects and their dependencies. "BusService" is the superspect of "IncomingCall". Advice in "IncomingCall" are now able to adapt pointcuts from "BusService" and to define advice dependencies to advice in "BusService".

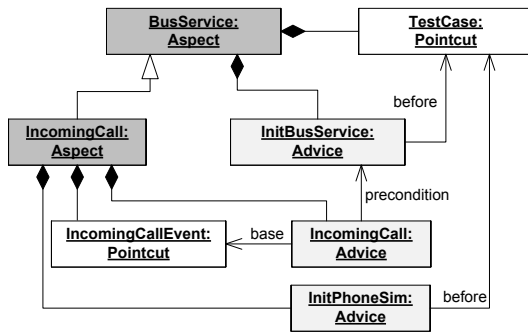


Figure 11. Aspect and advice dependencies.

### 4.3 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 2). The test focus is a set of aspects. There are three different weaving scenarios:

**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 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 emit their 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 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 woven 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.

## 5. 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 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 3. Text test aspect.

The test trigger definitions are shown in Listing 4. The system is stimulated using a joystick, where for each joystick 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 "TextTest" in Listing 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 the joystick 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 joystick. This test selection criterion corresponds to the pointcuts in the joystick aspect in Listing 4.

```

Aspect InputMapping{
  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 JoystickEnter base Enter <<
    JoystickService.Enter() >>
}

```

Listing 4. Joystick aspect.

When we combine the pointcut definitions of the “Joystick” 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 “Joystick”. 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.

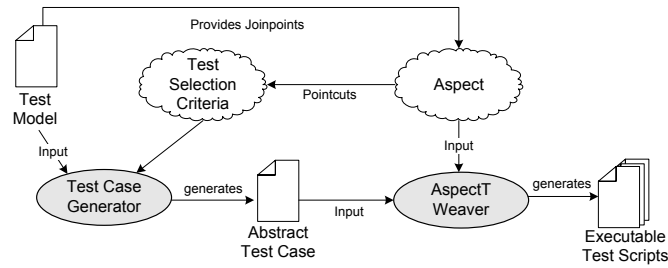


Figure 12. Aspect-oriented test case generation.

Figure 12 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.

## 6. Case Study

In this section we introduce a small case study for the application of AspectT in the automotive infotainment domain. The case study is a part of a graphical user interface (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

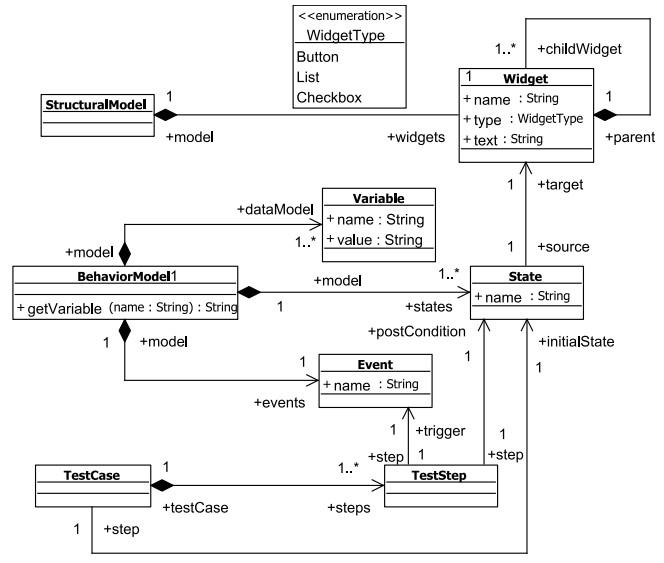


Figure 13. Part of the case study’s metamodel.

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 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 1. First we define a basic set of aspects to trigger the SUT by joystick 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 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 5.** The bus service configuration.

In the case study the events are triggered by a driver component that sends the corresponding bus messages. Listing 6 shows the aspect definition for the input mapping. The joystick service relies on the bus service. Therefore the “JoystickEnter” advice has the inter advice dependency *precondition* to the advice “InitBusService”.

```

Aspect Joystick extends BusService{
  PointCut Enter{
    Event : self.name = 'Enter'
  }
  Advice JoystickEnter base Enter precondition InitBusService
  <<JoystickService.Enter() >>
  //...
}

```

**Listing 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 OCR software. The aspect that implements the corresponding test oracle is shown in Listing 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.** 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 in a list using the joystick. 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 8.** Testing dynamic list behavior.

The aspect in Listing 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 precondition 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 9.** Timing test.

The next example demonstrates the behavior variation using AspectT. The pointcut condition is fulfilled for each state that has

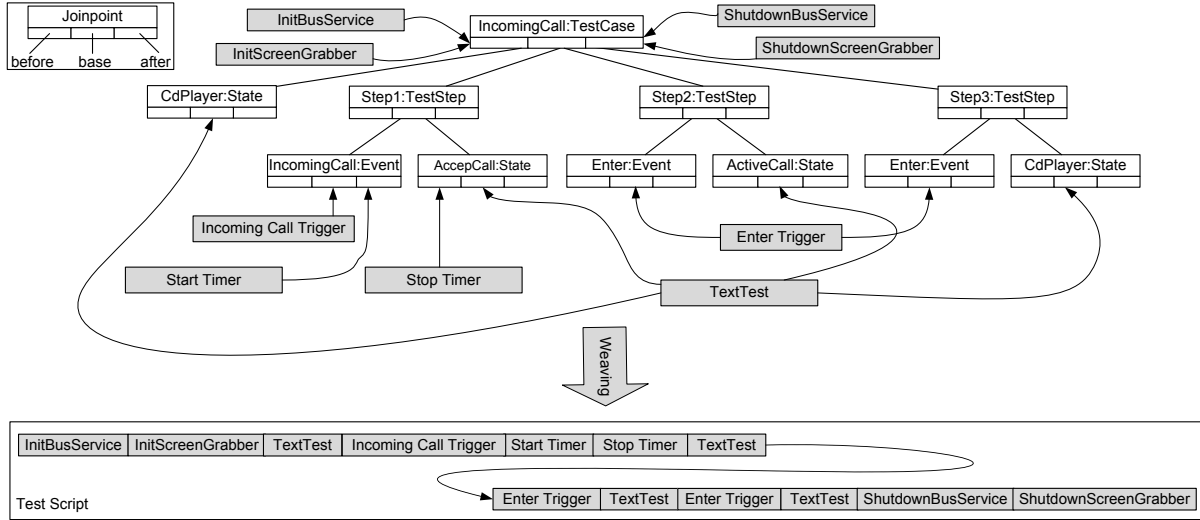


Figure 14. AspectT weaving.

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()
JoystickService.triggerLeft()
if (text != ScreenGrabberService.getText()){
  Error.log('<%=self.name%> Text changed on Joystick left')
}>>
}

```

Listing 10. Example for behavior variation.

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 “Joystick” 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 “Joystick” 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 joystick inputs that trigger no transition in a certain state. The test focus is defined by the aspects “BehaviorVariation”, “IncomingCall” and “Joystick”.

- **Address book test:** This test focus tests the dynamic address book behavior which requires the aspects “Joystick” and “AddressBookTextTest”.

Figure 14 shows the weaving of the *Text and Timing* test focus into an abstract test case and the resulting test script.

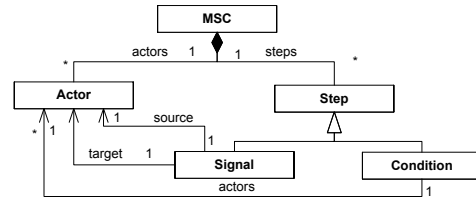


Figure 15. MSC based test case.

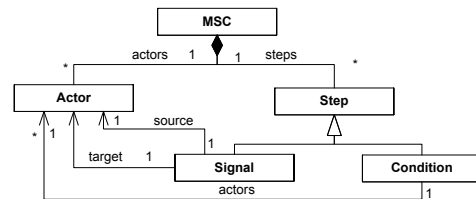


Figure 16. A simple MSC metamodel.

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 following we will demonstrate how it is possible to reuse the introduced aspects to instantiate an MSC based test case. Figure 15 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 a joystick 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 “Joystick” aspect to simulate the environment inputs. In order to achieve this, we have to adapt the existing pointcuts to the MSC metamodel. Figure 16 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 11 shows the extended pointcuts and the bus message aspect.

```

Aspect IncomingCallTrigger {
  PointCut IncomingCall{
    Event : self.name = 'Incoming_Call'
    Signal : self.name = 'IncomingCall'
  }
  ...
}
Aspect Joystick 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 11.** Extended pointcuts.

## 7. Results

In Figure 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 test framework implements the approach we presented in [4]. 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 of specification and test models.

Originally the test framework developer had to implement the test script generation for each test focus. After applying the AspectT framework, the test script generation could be separated from the framework and each test engineer could define his 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.

A further result was that the test case generation is more and more used for creating the preamble of a test case. 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.

## 8. Related Work

Extensive research has been performed in the area of model-based testing. The main focus of these studies lies 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 [5], for example, a transformation approach that generates test scripts for test cases that are created from B [2] 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 [3] and TGV [11]. 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<sup>9</sup> or XPAND2<sup>10</sup>. 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 in 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 [13, 15, 7] many modularization approaches that divide an attribute grammar into separate modules exist, where each module can be reused in different programming language compilers to

<sup>9</sup> <http://www.eclipse.org/modeling/m2t/?project=jet#jet>

<sup>10</sup> <http://www.openarchitectureware.org/>

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.

In [9], Gray et al. use aspect-oriented concepts to perform model evolution. Their approach is based on a metamodeling framework similar to EMF: the Generic Modeling Environment (GME).<sup>11</sup> 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 [18]: 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.

## 9. 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 paper 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 Eclipse Modeling Framework (EMF), where every EMF model is a potential join point model. This enables the application of AspectT to any modeling 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.

## Acknowledgments

I would like to thank Reinhard Stolle for his advice and help. I also thank Elmar Juergens and Eric Eide for their helpful comments on this paper.

## References

- [1] Message sequence charts. ITU-T recommendation Z.120, 1996.
- [2] J. R. Abrial. *The B-Book. Assigning programs to meaning*. Cambridge University Press, 1996.
- [3] A. Belinfante, J. Feenstra, R. G. de Vries, J. Tretmans, N. Goga, L. M. G. Feijs, S. Mauw, and L. 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.
- [4] S. 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.
- [5] F. Bouquet and B. 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, Sept. 2003.
- [6] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.
- [7] O. de Moor, S. L. P. Jones, and E. V. 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.
- [8] J. Grabowski, D. Hogrefe, G. Rthy, I. Schieferdecker, A. Wiles, and C. Willcock. An introduction into the testing and test control notation (TTCN-3). *Computer Networks, Volume 42, Issue 3*, pages 375–403, June 2003.
- [9] J. Gray, Y. Lin, and J. Zhang. Automating change evolution in model-driven engineering. *Computer*, 39(2):51, 2006.
- [10] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [11] C. Jard and T. Jérón. 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.
- [12] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.
- [13] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
- [14] Object Management Group (OMG). Meta object facility (MOF) specification. formal/2002-04-03, April 2002.
- [15] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995.
- [16] W. Prenninger, M. El-Ramly, and M. Horstmann. Case studies. In *Model-Based Testing of Reactive Systems*, pages 439–461, 2004.
- [17] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.
- [18] 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.

<sup>11</sup> <http://www.isis.vanderbilt.edu/projects/gme/>