Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-MSC-2009-17

Masterarbeit

im Studiengang "Angewandte Informatik"

Test Case Generation using Model Transformations

Dennis Neumann

am Institut für

Informatik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

6. November 2009

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel.	+49 (5 51) 39-17 20 10
Fax	+49 (5 51) 39-1 46 93
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 6. November 2009

Master’s thesis

Test Case Generation using Model Transformations

Dennis Neumann

November 6, 2009

Supervised by Prof. Dr. Grabowski
Software Engineering for Distributed Systems Group
Institute for Informatics
Georg-August-University Göttingen

Abstract

Despite a lot of effort in recent research, testing of software systems is still performed manually most of the time without taking the advantages of Model Driven Development. Such advantages include improved exploitation of models from early design phases for later development phases, easier adaptation when changes become necessary, or less effort for ports to other target platforms. A solution for this problem is proposed by a methodology called Model Driven Testing. In this thesis, an approach is presented which realizes the ideas of this methodology and the possibilities of automatic test case generation from models that are defined using the UML Testing Profile (UTP) are investigated. To reduce the complexity of UTP, a format for an intermediate model called SWETest is proposed, along with the corresponding model to model transformation rules from a constrained UTP variant where test behaviors are specified in interactions. From this intermediate model, code generation for TTCN-3 as target language is defined and implemented. Further model to model transformation definitions allow the generation of deterministic behaviors from ambiguous interactions.

Keywords: Model Driven Testing, Unified Modeling Language, UML Testing Profile, Model to Model Transformations, Test Case Generation, TTCN-3

Contents

1 Introduction

Modern software systems are getting more complex every day. To reduce this complexity, the use of models was introduced into the development of software. Model driven development promises a better quality of the end product, easier maintenance, reusability on different platforms, and automatic generation of the system structure or even of the entire system. Today, the most common technology used for modeling is the Unified Modeling Language (UML) [38].

While it is common practice to build a model of the system at least for documentation purposes, tests for the system are in most cases still developed without any support of models. Since up to fifty percent of the resources during software development are spent on testing [30] and as test suites are also programs, models should be considered in the construction of tests, as well. Currently, there is a lot of research on bringing models into the test development, but there is no universally accepted method yet.

This thesis is mainly based on the ideas and methods proposed by Baker et al. [17] and Zander et al.[45] The goal is to use the UML Testing Profile (UTP) – an extension of UML for testing – to model tests and map the models to executable test cases. UTP mainly consists of *stereotypes* which can be applied to ordinary UML elements. A stereotyped element gets a special meaning in the context of the used profile. For example, a UML class can become a UTP test component. In this thesis, a method to automatically generate test cases from UTP models is presented. The method is implemented with openArchitectureWare [12], a model transformation toolset for the Eclipse platform [5].

UTP, being a UML profile, contains several constraints which are however mainly applied to its stereotypes. This means that all UML concepts and metaclasses can be used in a UTP model. Therefore, a UTP model can be potentially very complex and hard to process automatically. In this thesis, constraints for UTP models were created that restrict the structure of a model. This way, it is possible to analyze the models programmatically and the needed data can be extracted easier.

The constraints on UTP are the first step in simplifying and formalizing the test model. In the second step, a UTP model is transformed into a model that is structured in a precise way. This means that all the elements are placed in exactly defined containers and relate to each other in an exactly specified way. To achieve this, a new metamodel named SWETest is presented that contains only the relevant test elements. It conforms to the EMF [42] implementation of EMOF [34] called Ecore. The naming and the structure of the SWETest elements are inspired by the UML metaclasses and UTP stereotypes. SWETest can therefore be viewed as a greatly simplified, strict, and more structured version of UTP. The main advantage of SWETest is that once a UTP model is transformed into an SWETest model, it can be used in a much easier way to generate test cases for several different languages. As an example, the testing language TTCN-3 [16] has been chosen to be the target language for the generation of test cases. Additionally, an SWETest model can be used for further model to model transformations. An example is also provided in this thesis. In this example, the statements in test cases involving distributed test components are sorted topologically so that deterministic test case behavior can be produced. Summarized, the contribution of this thesis is as follows:

- By constraining UTP, a reasonable subset of it has been chosen which is easier to process programmatically.
- An intermediate Ecore metamodel called SWETest has been developed into which UTP models can be transformed. It is structured strictly and can be used as the foundation for the generation of code for arbitrary programming languages or testing languages.
- Additional model to model transformations have been defined to generate deterministic test cases from ambiguous interactions.
- A mapping from the intermediate SWETest metamodel to TTCN-3 has been defined and implemented.

The thesis is structured as described in the following. In Chapter 2, the used concepts and tools are briefly described. General concepts of software testing are presented. Besides the modeling technologies UML and UTP, the existing methodologies for model driven development are discussed. Next, an algorithm for topological sorting is described which is used to produce several deterministic test cases from a non-deterministic test case. The two tool collections Eclipse Modeling Framework (EMF) and openArchitectureWare (oAW), as well as the testing language TTCN-3 are presented in the subsequent sections. Finally, the work in this thesis is compared to some related methods of test case creation using models.

The description of the test case generation method created in this thesis can be found in Chapter 3. First, the constrained version of UTP, called SWEUTP, and the Ecore metamodel that is used for validating transformed models, are presented. After that, the model to model transformations as well as the model to text transformation are defined and explained.

An overview of how the constraining and the transformations have been implemented is given in Chapter 4. The used tools are referred to and their interaction is explained. Some exemplary code fractions are presented for each transformation.

In Chapter 5, a complete example of a test case scenario including its transformation and test case generation is presented. The example involves the academic communication protocol called Inres. Test cases for a hypothetical implementation of Inres are created as UTP models, transformed into an SWETest model, and then transformed into TTCN-3 code. Finally, Chapter 6 concludes with a short summary of the work and an outlook.

2 Foundations

The theoretical basics to the work of this thesis are given in this chapter. First, the general software testing techniques are mentioned. Then, the relevant UML concepts and diagrams are described, as well as UTP. The modeling paradigms Model Driven Engineering (MDE) and Model Driven Architecture (MDA) for building systems using models, as well as Model Based Testing (MBT) and Model Driven Testing (MDT) which are used for creating tests using models, are explained next. In this thesis, the transformations and the code generations are performed with the help of several plug-ins of the Eclipse platform. The two most important ones, the Eclipse Modeling Framework (EMF) and openArchitectureWare, are presented in this chapter. Also, the target testing language called Testing and Test Control Notation Version 3 (TTCN-3), for which the test cases are generated, is described briefly. Finally, some publications and tools are presented that contain concepts similar to the method developed in this thesis.

2.1 Software Testing

Myers [30] describes software testing as "the process of executing a program with the intent of finding errors". It is assumed that any program of a reasonable size cannot be perfect, which means that it contains errors. Therefore, the goal of good software testing is to find and correct as many errors as possible. However, even for a simple program, it can be computationally impossible to find all errors. This means that a successful testing of software depends on the experience of the tester and the choice of the right testing strategy.

One important software testing kind is black-box testing. This strategy concentrates on the input and output data of the tested program at its interface level. The tester tries to execute the program with as many different input data sets as possible and observes the output. The inner structure of the program is hidden and does not influence the choice of the input data.

Another testing strategy kind is white-box testing. Using this method, the tester examines the code of the tested program in order to derive test cases from it. The goal is to execute the program with the right arguments, so that as much behavior is covered as possible. Strategies for covering behavior include statement coverage, condition coverage, branch coverage, and path coverage.

Communication protocols can be tested with the method called *conformance testing*. It checks if a protocol has the correct behavior as defined in its specification. Holzmann [26] differentiates between two kinds of conformance testing for protocols. For the *functional* conformance testing, it is sufficient that the specification of the tested protocol is defined informally, for instance, as a written document. The input and the output data values are derived from the specification. The main idea is to execute the protocol with as many input values as possible and check if the output values correspond to those derived from the specification.

The second kind of conformance testing described by Holzmann is the *structural* conformance testing. The emphasis of this method is on the control structure of the protocol. The implementation of the tested protocol and its specification must be representable as finite state machines containing states and transitions. During the test, the implementation is put into all possible states and input signals that cause the change of the state are sent to it. By verifying that the state was changed in the implementation exactly as in the specification, it is proven that the implementation is capable to reproduce the behavior defined in the specification.

2.2 Unified Modeling Language

Unified Modeling Language (UML) [38] is standardized by the Object Management Group (OMG) [11]. UML defines modeling concepts and diagram types enabling system developers to create models in order to design, analyze, and implement their systems. Additionally, the goal of UML is to make the models exchangeable between different tools and developers. To achieve this, UML is formally defined by a metamodel and there is an XML-based format, the XML Metadata Interchange (XMI) [36] format, that is used to ensure the interoperability between tools.

UML Metamodel

The OMG specified a four-layer architecture for metamodeling [37]. The UML metamodel and model levels are layers in this architecture and are called M2- and M1-level, respectively. Instances of model elements belong to a lower level, the M0-level. Those instances are run-time objects that are executed on a concrete system. The top level, M3-level, defines the structure of the UML metamodel, so it can be called meta-metamodel. It is called Meta-Object Facility (MOF) and it reuses parts of the UML syntax and semantics. Basically, the part for defining class diagrams is reused, since the UML metamodel itself consists of classes and relations between them. However, the MOF is a stand-alone metamodeling concept, which can be used for defining not only UML, but also other models or metamodels. Examples and detailed information can be found in the OMG specification of MOF [34].

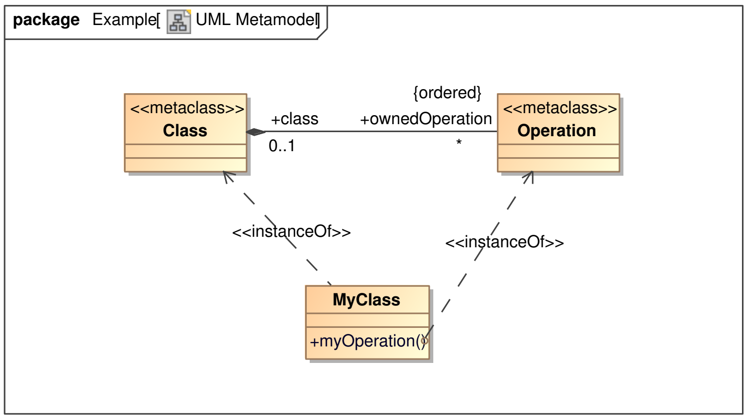


Figure 2.1: An example for the use of the metamodeling principle in UML

The MOF specification defines a metamodel as "a model used to model modeling itself". This means that a metamodel describes the rules for constructing correct models. The UML metamodel (M2-level) describes the allowed structure for UML models, which are defined by users.

A simple example for using a metamodel is presented in Figure 2.1. The two *metaclasses* `Class` and `Operation` belong to the UML metamodel. They define the according elements which can be used by a UML modeler to build system models. The composition arrow between the metaclasses means that operations can be parts of a class, namely an arbitrary number of operations can belong to zero or one class. The names of the association ends (in this case `class` and `ownedOperation`) specify navigation paths which, for example, can be used in Object Constraint Language (OCL) expressions. The additional keyword `ordered` means that the operations of a class are ordered alphabetically. The two *model* elements `MyClass` and `myOperation` are *instances* of the metaclasses. They are structured as specified by the metaclasses.

In Figure 2.1, UML *keywords* are shown[1]. Keywords are used to distinguish different UML elements from each other, which use the same graphical representation. A keyword is placed above or next to the name of an element and is enclosed in guillemets. In this example, the keyword `metaclass` is used to indicate that the elements are not simple `Class` elements, but metaclasses defined in the metamodel. The full list of UML keywords can be found in the UML Superstructure Specification [38].

[1]'instanceOf' is not a standardized keyword. However, it is used in the UML specification in similar examples, so this notation is also used here

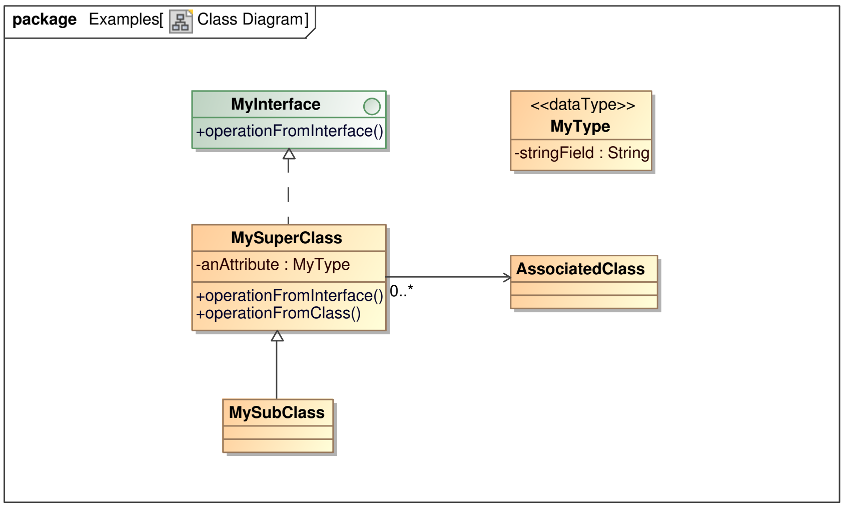


Figure 2.2: Elements of a UML class diagram

UML Modeling Concepts

One of the main terms in UML is *model*. It consists of elements, which together form a view of a system. The elements can be, for example, classes, data types, attributes, or states. To have a better overview of the model, some or all elements can be put together to diagrams that show different views of the underlying model. One diagram does not necessarily show all aspects of a modeled system. UML rather offers different diagrams for displaying different aspects of a system. There are, for example, diagrams that show the structure of the system or its components and other diagrams that display behavioral features. The UML diagrams that are relevant for this thesis are described briefly in the subsequent sections. The relevant sections of the underlying metamodels for the elements in class diagrams, sequence diagrams, activity diagrams, and composite structure diagrams can be found in Appendix A.

Class Diagrams

A class diagram shows the static structure of a system. It displays single parts of a system as well as relationships between those parts. Some elements that can be contained in a class diagram are shown in Figure 2.2. A general term for classes, interfaces, data types, etc., is *classifier*.

- A *class* specifies a set of objects that have the same structure and behavior. Classes can have relations to each other. Two examples are shown in Figure 2.2: Inheritance relation depicted as an arrow with an empty arrowhead and directed association relation illustrated as an arrow with an open arrowhead. The multiplicity statement '0..*' means that an object of type `MySuperClass` can be associated with zero or more objects of type `AssociatedClass`.

- An *interface* represents a set of abstract public features. A class can realize an interface, which means that the class must provide implementations for all operations of the interface. In UML, an interface realization is depicted as a dashed arrow. Note how the operation from `MyInterface` is also contained in the class `MySuperClass`. The operation in the class can be implemented in a behavior (see subsequent sections).

- A `DataType` element is used to create user-defined types. Data types can have attributes of any other type, which means that nested data types of any depth can be modeled. The data types can be used in class definitions like any other type.

- *Signals* are used to send data between objects in diagrams which model the behavior of a system.

Sequence Diagrams

Sequence diagrams are used to display the behavior of a system. The focus is on the data exchange between systems or system components (Figure 2.3). Sequence diagrams show some or all of the elements of an interaction, which is contained in the model. An interaction is a container for elements that can be shown in a sequence diagram.

The main notation elements of a sequence diagram are lifelines. A lifeline always represents one participant in the information exchange modeled by the diagram. Such a participant can be, for example, a class object, a port, or a part (see also 'Composite Structure Diagrams' below). In the case of a multivalued part, the semantics is that one value of the part will be chosen arbitrarily at runtime.

The lifelines in the sequence diagram can exchange data either as messages or by calling operations on each other. A message can refer to a signal, which must be defined in the model. The signal is transmitted with the message to the receiver lifeline and can trigger a reaction. If the signal definition contains attributes, then the message may contain arguments which represent values for the attributes. Those values can be processed by the receiver. Signal messages are sent in an asynchronous way, which means that the sender continues its execution after sending the message. The other kind of messages, operation call messages, can be processed synchronously. In this case, the sender will wait for a reply message from the receiver before it proceeds with its execution. Call messages refer to operations which must be contained in the receiver object. A lifeline can also call an operation on itself (message 3 in Figure 2.3).

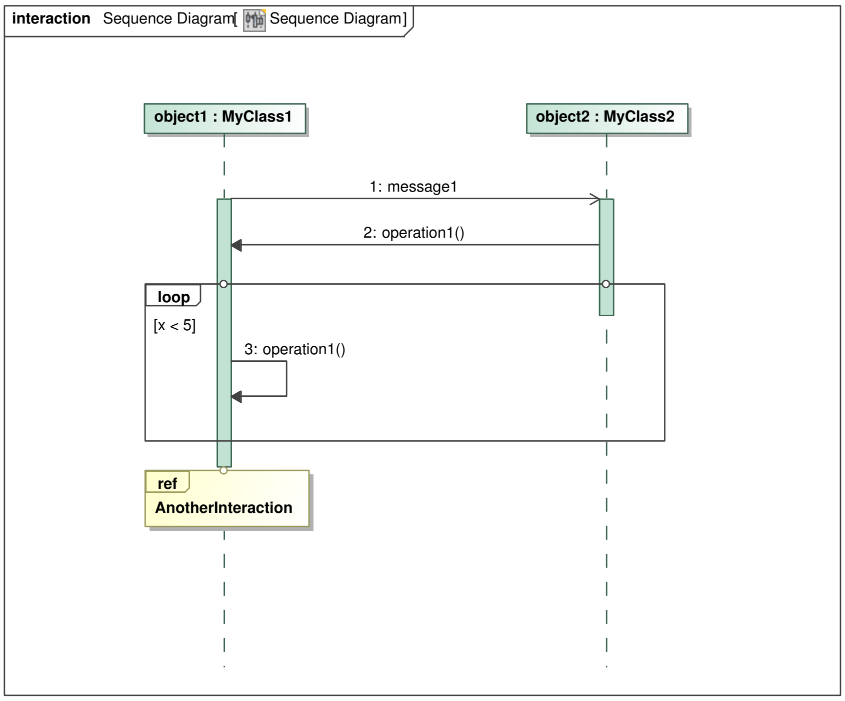


Figure 2.3: Elements of a UML sequence diagram

So called combined fragments are used to model different execution flows, for example:

- `loop` to construct loops which will execute as often as the given condition is true.
- `alt` containing several blocks with optional conditions to make alternative executions possible, similar to `if`-statements in general-purpose programming languages.
- `opt` (optional), which is similar to `alt`, but contains only one execution block.
- `par` (parallel) to model execution threads that should be processed simultaneously.
- `neg` (negative) for traces that are invalid.

A sequence diagram can contain references to interactions from other sequence diagrams ('AnotherInteraction' in Figure 2.3). The element is called `InteractionUse`. The referenced interaction will be processed in the place of the reference. After that, the execution in the calling interaction moves on.

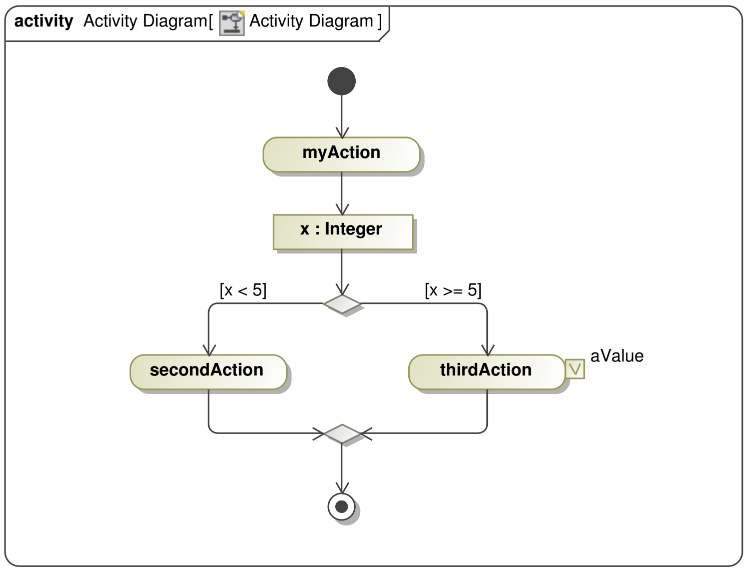


Figure 2.4: A simple activity diagram

A sequence diagram can model not only one execution sequence ('trace'), but rather a number of sequences. Sending and receiving a message generates events. The order of the events in an interaction defines a number of traces determined by the following two rules:

- A receive event happens always after the send event of the same message.
- An event happens after all events that are above it on the same lifeline.

Activity Diagrams

Like sequence diagrams, activity diagrams display specifications of behaviors. The main focus of this diagram type is on modeling sequences of actions. The container that holds the actions and other supporting elements in the model is called *activity*.

Actions and other elements that can be displayed in an activity diagram are modeled as nodes in a graph which are connected by edges modeled as arrows. Edges define the execution order of the actions and the occurrence order of other nodes in the activity. In Figure 2.4, a simple activity with three actions is shown. Actions can refer to behaviors (as `CallBehaviorActions`) or to operations (as `CallOperationActions`) which are defined elsewhere in the model. Action nodes can have input and output parameters, modeled as object nodes, which precede or follow the action in the graph. In Figure 2.4, the object node 'x' is an output parameter of the action 'myAction'. The value of such a node can then be

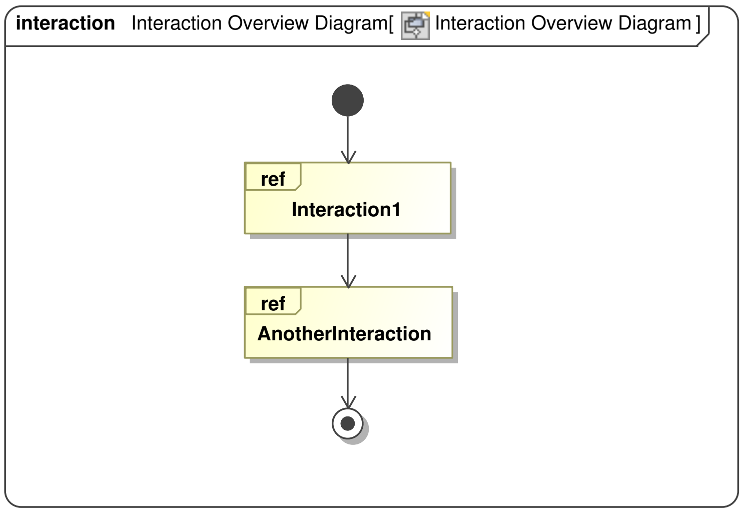

interaction Interaction Overview Diagram[Interaction Overview Diagram]
ref
Interaction1
ref
AnotherInteraction

Figure 2.5: A simple interaction overview diagram

connected to another action as an input parameter or, as in this case, be used in a decision node to determine which path in the activity graph is followed at runtime. Decision nodes and merge nodes are modeled as diamond symbols and are used to model control flows. The outgoing edges of decision nodes can have guard expressions which define conditions for the next edge to follow in the graph.

Another way to model a parameter of an action is a pin. There are input pins and output pins that are displayed as little squares attached to the action in the place where an edge is incoming or outgoing. In Figure 2.4, a third kind of pins is displayed, a value pin named 'aValue'. This pin represents an input to the action, which is not provided by object flow of the activity, but rather by evaluating a value specification that is included in the pin.

The two node types `InitialNode` and `ActivityFinalNode` are used to indicate the start and finish of an activity. Such nodes are displayed as a black circle and a black and white circle, respectively.

Interaction Overview Diagrams

An interaction overview diagram shows the interplay of interactions. This diagram type is an extension of the activity diagram. In addition to the elements of the activity diagram, `InteractionUse` nodes can be used. Those nodes represent references to interactions in the model. In Figure 2.5, one interaction is called before the other one. More complex examples could contain, e.g., decision and merge nodes as shown in Figure 2.4.

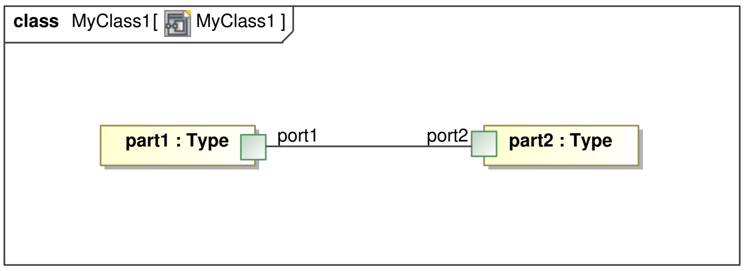


Figure 2.6: A simple composite structure diagram

Composite Structure Diagrams

The structure of a classifier can be illustrated in a composite structure diagram. The classifier consists of elements called *parts*, which are interconnected by connectors (Figure 2.6). Parts define the runtime configuration of the classifier they belong to. A part is an object or a set of objects, which have a type and are instantiated with the creation of the classifier. Connected parts can exchange data through the connectors. Additionally, the parts can have ports that serve as interfaces to the respective connectors and encapsulate the behavior of the parts.

Object Constraint Language (OCL)

Some specific system restrictions cannot be described in UML. The objects of the modeled system are often described insufficiently by the model alone and need to have additional constraints that must always hold. For this reason OCL was developed by the OMG [35]. Statements in this constraint and query language can help making a model more restrictive and more formal. OCL statements are free of side effects, which means that they never change the underlying UML model. OCL defines several types of constraints and queries, which are described in the following list.

- Initial values can be assigned to attributes and association ends of a classifier (`init`).
- Similarly, a value can be derived from other values queried in the model. For example, two integers can be multiplied and assigned to an attribute (`derive`).
- A body can be defined for an operation (`body`).
- Besides the attributes and operations contained in the model, new ones can be defined with OCL (`def`). However, those must be derived from existing ones.

- An operation can get preconditions and postconditions. These kinds of constraints prescribe with a boolean expression what must hold before and after an operation call, respectively (`pre` and `post`).
- An invariant is a boolean expression that must always evaluate to true (`inv`).

An OCL expression consists of the specification of the assigned UML element (the context), the type of the constraint and the actual constraint or query. The type must be one of the abbreviations described above. The following definition describes the general structure of an OCL expression.

```
context <element name>
<type>: <evaluated expression>
```

Optionally, the constraint can have a name that must follow the type (see examples below). The actual evaluated expression can be either a boolean expression (i.e., a constraint) or a query which returns one or more elements from the model. Both in constraints and queries, the dot operator '.' can be used to navigate through the model, e.g., via associations, or to get to the contained attributes and operations. The arrow operator '->' must be used if a predefined OCL operation is called. Some of the OCL operations used in this thesis are the following:

- `select` returns all elements of a collection for which a given boolean expression is true;
- `exists` returns a boolean value that indicates if a certain element is present in a collection;
- `forAll` checks if a condition is true for each of the elements in a collection;
- `any` returns one element from a collection for which a condition is true;
- `notEmpty` verifies that a collection contains elements.

An OCL constraint can either be assigned to a user-defined element in a model (M1-level) or to a metaclass in the UML metamodel (M2-level). The following two examples illustrate the differences between these two sorts of constraints.

```
context Person
inv positiveAge: self.age > 0
```

In this example, it is assumed that a user-defined class named `Person` exists in the UML model and contains the attribute `age`. The invariant named `positiveAge` is assigned to that class. It states that the `age` attribute must always be greater than zero.

```
context Class
inv classId: Class.allInstances().ownedAttribute
                                ->exists(attr|attr.name='id')
```

Here, the `Class` element from the UML *metamodel* is targeted. This means that the constraint is assigned to all the classes in the UML model, no matter what their name is. The invariant will only evaluate to true if all the classes in the model contain an attribute named `id`. The operation `allInstances` is defined in the OCL and returns all the instances of a specified type.

UML Profiles

For some modeling purposes or usage in a specific domain, UML by itself is not precise or flexible enough. For this reason, UML offers extension mechanisms for the definition of domain-specific languages. Changing the UML metamodel or even building a new metamodel on MOF is a heavy-weight extension mechanism which allows a modeler to define new metamodels with a different syntax and semantics. This method does not set any limits to the modelers to design their own modeling languages. However, it is possible that such languages cannot be processed by standard UML tools. In contrast to that, defining a UML profile is considered a light-weight extension mechanism. Profiles are specialized separate packages that do not change the UML syntax, but rather define extensions and restrictions to the standard UML language. The restrictions must not be contradictory to the UML syntax or to the existing restrictions.

A profile includes *stereotypes* for extending and *constraints* for restricting the UML metamodel. Constraints are expressed in OCL. Stereotypes are defined in the UML metamodel layer (M2) and can be viewed as labels that can be defined as applicable to an arbitrary UML element (but not to another stereotype). If a stereotype is applied to an element, the structure of the element remains the same, but the element gets a special meaning. A simple example is shown in Figure 2.7. On the left, a stereotype with the name `JavaClass` is defined inside a profile. As indicated by a filled arrow, the stereotype extends the `Classifier` meta-element from the UML metamodel. This means that the defined stereotype can be applied to all model elements of type `Classifier` and its descendants (including for example `Class`). On the right, a class is defined in the UML model layer (M1) and the stereotype `JavaClass` is applied to it. The syntax is identical to UML keywords, so stereotypes cannot have the same name as a keyword.

Stereotypes can have attributes which become *tagged values* in the stereotyped elements. Those values can be used for storing additional information about the modeled element. Additionally, profiles may contain model elements (as opposed to metamodel elements) like interfaces or enumerations.

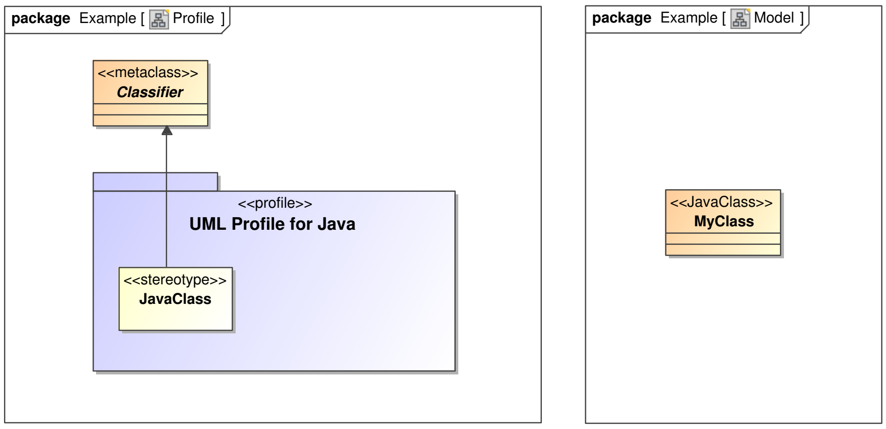


Figure 2.7: Definition and usage of a stereotype

2.3 UML Testing Profile

The UML Testing Profile (UTP) [33] extends UML with elements that are useful for testing purposes. It can be used to model artifacts of test systems like test components and test cases. UTP is divided into four groups of concepts which are described in the following subsections.

Test Architecture Concepts

In this concept group, structural elements are defined that are necessary to design tests. These elements communicate with each other and with the tested system, the System Under Test (SUT). In Figure 2.8, the definitions of the elements in the profile are shown in a class diagram. Additionally, the corresponding extensions and the extended metaclasses from the UML metamodel are depicted.

The *test component* is defined as a stereotype for a `StructuredClassifier`, which is an abstract metaclass and an ancestor of the `Class` metaclass[2]. Test components participate in test behaviors by communicating with each other and with the SUT. These interactions define the test cases. The `SUT` stereotype can only be applied to a property of a classifier. This can be, for example, a class attribute or a part in a composite structure diagram. This property is a reference to the SUT, which can be imported from another project or defined as one or more classifiers in the same project. The SUT can consist of one or several elements that can communicate with each other and with the test components.

[2]In this thesis, only classes are being extended by the TestComponent and TestContext stereotypes. It is also possible to use other structured classifiers like collaborations.

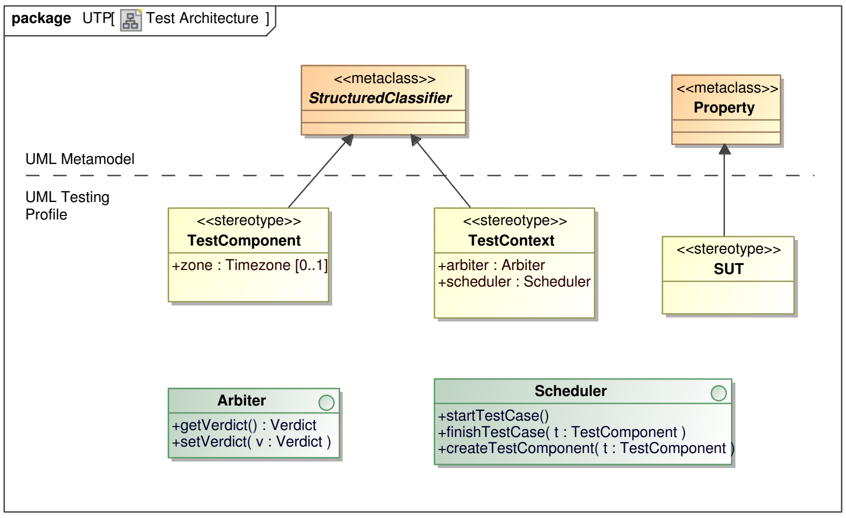


Figure 2.8: UTP test architecture elements in a class diagram

The *test context* serves as a container for other test elements. It is modeled as a structured classifier with the stereotype `TestContext`. It contains one or more *test cases* which are represented as operations with the stereotype `TestCase` (see below). The composite structure of the test context classifier, which is called the *test configuration*, shows the static structure of the communication between the test components and the SUT. In addition to the mentioned stereotypes, there are two interfaces that are defined in the test architecture: `Scheduler` and `Arbiter`. Realizations of these interfaces must be included in the test context as properties[3], exactly one of each. The scheduler creates, controls, and destructs test components. It starts and stops test cases and it contains the information about which component belongs to which test case. At the end of the test, the scheduler informs the arbiter to set the final verdict. The task of the arbiter is to provide the statement at the end of a test, if the test has been successful or not, if for some reason the result cannot be stated, or if some other error has occurred.

In Figure 2.9, an example of a test architecture model is shown. Note that the type of the SUT is not stereotyped as it is a classifier that could be imported from another package or project. The SUT and the test component are used as parts in the composite structure (test configuration) of the test context (Figure 2.10). The test configuration defines that the test component can stimulate the SUT by sending messages or calling operations through the port `myPort2`. Answers sent by the SUT are forwarded to `myPort1`, so that the test component is able read them.

[3]In Figure 2.8 arbiter and scheduler are tag definitions of the `TestContext` stereotype. However, the UTP specification [33] also states that they must be properties of the stereotyped classifier, i.e., not tagged values.

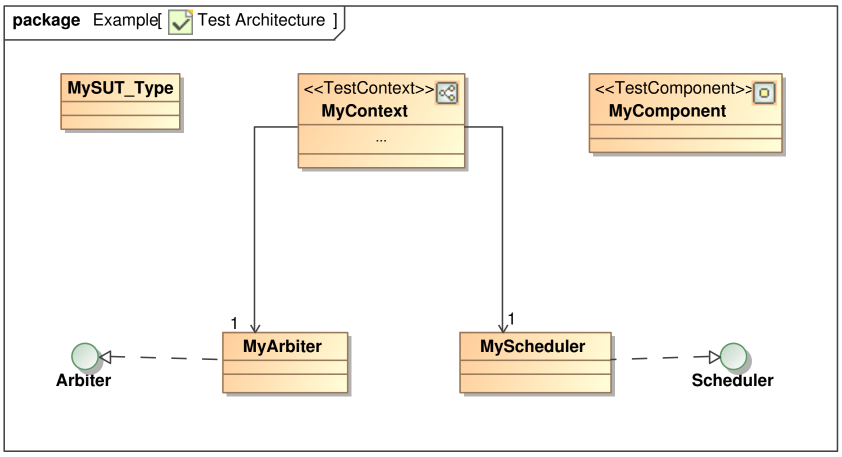


Figure 2.9: UTP test architecture example

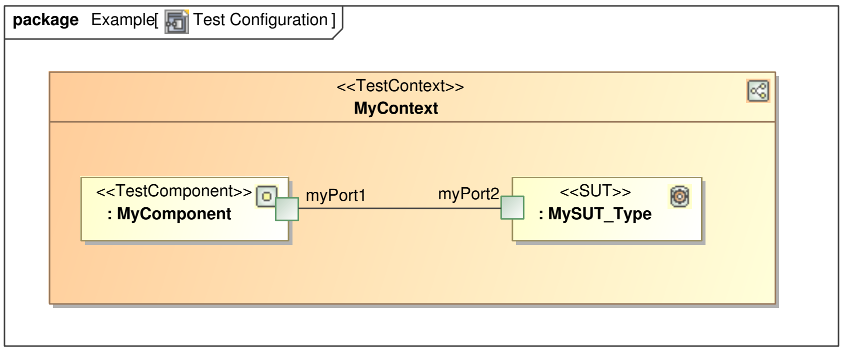


Figure 2.10: UTP test configuration example

Test Behavior Concepts

This part of UTP contains stereotypes and enumerations which are used to specify the behavior of tests. Test cases are modeled as public operations of the test context and marked with the stereotype `TestCase`. The operations refer to behaviors which can be for example UML interactions, activities, or state machines.

`TestObjective` is a stereotype for the metaclass `Dependency`. It is used to refer from a test case or a test context to an object that specifies the objective of the test (possibly just a hand-written text). Test cases realize test objectives by specifying the concrete behavior that is described in the objective. Since test cases are included in the test context, the SUT and the test components from the test configuration can be accessed directly. Test cases describe the interplay (i.e., data exchange) of those test elements. A test case returns a verdict value, which can be `pass`, `fail`, `inconclusive`, or `error`. These values are defined in the enumeration `Verdict`. If a test case returns the verdict `pass`, it means that the test has been executed as expected. The value `fail` means the opposite (for example, an unexpected message might have been received). An `inconclusive` is returned if it cannot be decided whether the test passed or failed. The value `error` is returned if problems occur within the test system itself.

Test case specifications normally define the expected behavior. In contrast to that, *defaults* are typically used to specify unexpected or erroneous situations and reactions to those situations. UTP contains the `Default` stereotype, which can be applied to behaviors. The default behaviors can be reused in different test cases. To make a default active, it must be referred to inside a test case specification. In an interaction, for example, a comment containing the name of the default can be attached to a lifeline.

UTP defines a number of stereotypes for actions. The `FinishAction` stops the execution of the test component that called the action. The `ValidationAction` implicitly sets a verdict for a test component, i.e., this action forwards a verdict to an arbiter. The `LogAction` indicates that some data about the test case should be logged. Other stereotypes on actions are described below in the 'Time Concepts' subsection.

Test Data Concepts

UTP defines data concepts that are specific for testing and are not contained in the standard UML specification:

- The `LiteralAny` wildcard value which can be used to indicate an arbitrary value. It is defined as a stereotype for the `LiteralSpecification` metaclass.
- The `LiteralAnyOrNull` value to indicate an arbitrary value or no value at all.
- A *data pool* contains sets of values called *data partitions*, which can be used for stimulating the SUT repeatedly or to define equivalence classes of values. A data pool can also contain individual values.

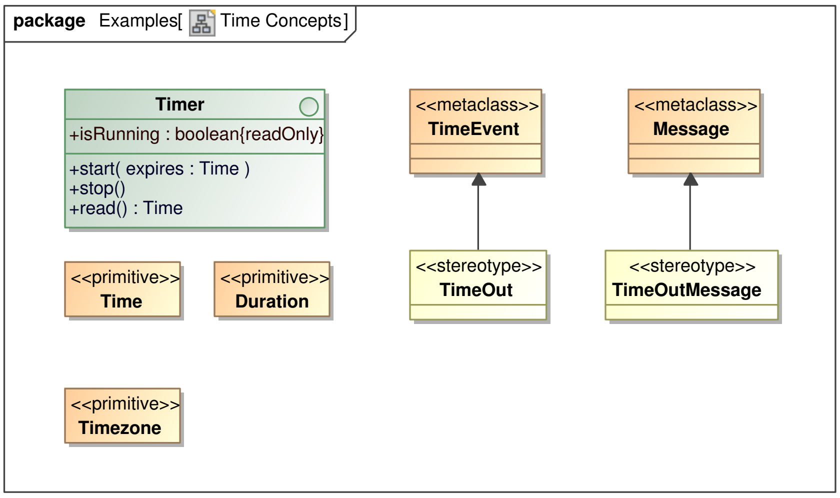


Figure 2.11: UTP time concepts

- *Data selectors* are operations in a data pool or a data partition. As the name suggests, they are used to select some data values out of a set of values.

Time Concepts

Timing events are important in the testing of distributed systems, because the connections between the system components are often unreliable and lost messages must be detected. For this reason, UTP contains extensions to the UML time concepts. Timers can be used to control the test case execution or to recognize lost messages with the help of timeouts. A timezone is a means to synchronize different test components with the same time.

`Timer` is a predefined interface which offers a typical timer attribute and operations (Figure 2.11). The primitive type `Time` is used to define a point in time. `Duration` can be used to specify a period of time. A timer is started with a predefined expiration time. When the timer expires, a timeout message is automatically sent to the owning class and a timeout event is generated. For this purpose, the `TimeOut` stereotype is used which can be applied to a `TimeEvent` metaclass[4].

For simpler use and control or timers and timeouts, some supportive actions are defined in the UTP with obvious functionalities:

- StartTimerAction,

[4]The UTP specification describes the application of TimeOut to the metaclass TimeTrigger, which however does not exist in the current UML specification

- StopTimerAction,
- ReadTimerAction,
- TimerRunningAction,
- TimeOutAction.

`TimeZone` is a primitive type which is used for synchronization of test components. Components having the same time zone value are considered to have the same time, and are thus synchronized with each other. Time zone values can only be compared for equality and are not ordered. The values can be set and manipulated by the two actions `GetTimeZoneAction` and `SetTimeZoneAction`.

2.4 MDE and MDA

In this section, two approaches to software development using models are described. The first one, which is called Model Driven Engineering (MDE), is also often referred to as Model Driven Software Development (MDSD). It is a general idea of creating models to implement a system or parts of a system instead of writing code. The second approach, Model Driven Architecture (MDA), can be viewed as a realization of MDE. MDA consists of concrete specifications and process descriptions that should be used for software development.

Model Driven Engineering (MDE)

The main concepts of MDE are models, metamodels, and transformations. According to Bezivin [19], the basic principle of MDE is that "Everything is a model", which is an extension of the principle "Everything is an object" from object orientation. He suggests two terms to describe the relations in MDE: *representation* and *conformance*. A real world system is represented by a model, whereas a model conforms to a metamodel. The software life cycle is viewed as a chain of model transformations. This means that there are models with varying abstraction levels that can be transformed into each other. In this context, even source code is viewed as a model and its syntax description as a metamodel.

Favre [22] defines MDE as "an open and integrative approach that embraces many other Technological Spaces in a uniform way". A Technological Space can be for example *Modelware* in which UML is used, *Dataware* with SQL, or *Documentware* with XML. In each of those spaces there exist the notions of a model (UML model, data in a database, an XML document) and of a metamodel (UML metamodel, a database schema, an XML schema).

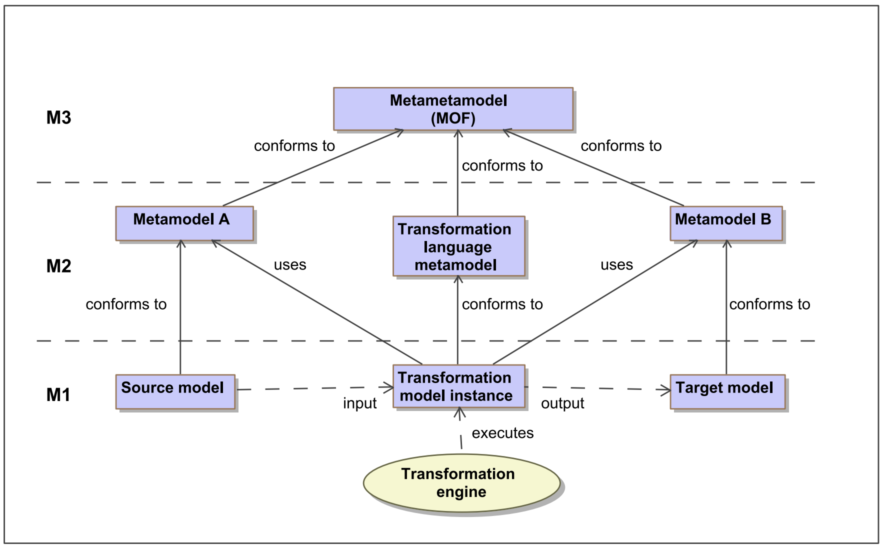


Figure 2.12: Model to model transformation process [23]

Model Driven Architecture (MDA)

MDA is a set of specifications by the OMG which together form "an approach to using models in software development"[32]. It is a framework that defines how models in one language can be transformed to models or source code in another language. The main idea of MDA is to separate platform-independent and platform-specific system specifications which are represented as models. The main advantages of MDA are:

- Models make the system easier to understand, because the system developer can get a better overview of the system. Hence, it is easier to develop bigger projects.
- For the same reason, the already developed system is easier to maintain.
- Parts of the system can be generated automatically. For example, interfaces to other systems or system parts can be generated, which makes the integration in a bigger system easier.

MDA defines different kinds of models to represent a system. The two most important ones are the Platform Independent Model (PIM) and the Platform Specific Model (PSM). The PIM has a high level of abstraction and does not contain any information about the concrete implementation technology. A PIM can be transformed to several PSMs, each one adapted to a certain implementation technology.

The general process of transforming a model into another model is illustrated in Figure 2.12. The transformation is performed by a transformation engine which executes a transformation model instance. This model instance contains definitions how an input model should be transformed to an output model. To achieve this, the transformation model instance has access to the metamodels of the source model and the target model. The model instance itself also has a metamodel which it must conform to. All the metamodels conform to a higher-level metametamodel. In MDA, the Meta Object Facility (MOF) is used as the metametamodel. Instances of MOF, i.e., metamodels, serve as definitions for the concrete user-defined models. MDA does not prescribe a specific metamodel, but suggests the UML metamodel or other metamodels derived from MOF. There also exists a specification for a transformation language metamodel called Query/View/Transformation (QVT) [39].

After the transformation is completed, the PSM contains all the needed data about the system and information about the targeted platform. In the next step, a model to code transformation is performed. Similar to a model to model transformation, a code generation engine reads the PSM and executes a model instance with definitions to generate source code.

2.5 MBT and MDT

Similar to methodologies using models in the development of software systems, there exist concepts for creating testing specification with models. The main idea of Model Based Testing (MBT) is to derive executable test cases from existing models of a system or a system specification. In contrast to that, the goal of Model Driven Testing (MDT) is to define test cases themselves as models and then to generate executable code using model to model and model to text transformations.

Model Based Testing (MBT)

The MBT methodology requires a specification of the System Under Test (SUT) in the form of a model to be present [18]. If this is the case, then tests can be created based on a graphical model that describes the behavior of the SUT. The idea is that a number of test cases are generated automatically by analyzing the system model. For example, there exist methods for deriving test cases from state automata [41] and from transition systems [20].

A *path* in a system model is a sequence of events that can occur in the system. This means that a path defines one possible execution scenario. At the beginning of the test case generation process, all the paths that should be tested are determined. The selection criterion can be, for example, path coverage or branch coverage [43]. In those cases, path coverage includes all the possible paths whereas branch coverage means choosing paths so that each branch is included at least once.

Each selected path is traversed and testing directives for each event in the path are generated. A testing directive can be, for example, a simulation of a user input or a call of an operation in the SUT. All the testing directives of a path form a test case. The generated test cases can be applied to the actual system that is represented by the analyzed model. During the execution, the system follows the same paths determined in the model and reacts to the stimuli of the test cases.

Model Driven Testing (MDT)

Baker et al. [17] define Model Driven Testing as application of the MDA principles to software testing. In contrast to MBT, in this approach the tests themselves are specified as models, independently from the specification of the system that is tested. Executable test cases are then generated through model to model and model to text transformations.

Similar to MDA, there are models defining tests on different abstraction levels [45]. The Platform Independent Test model (PIT) specifies tests on a high abstraction level, not relying on a specific technology. Those tests are transformed into the Platform Specific Test model (PST) through model to model transformations. This model contains details of the technology on which the tests will be executed. In the last step, the PST is transformed into executable code of a programming language.

2.6 Topological Sorting

As described previously, a UML sequence diagram defines interaction fragments that are partially ordered. This means that one diagram specifies a number of possible sequences of the fragments. In this section, an algorithm for retrieving all the possible sequences, i.e., *linear extensions* of a partially ordered set of elements is described.

A partial order $\leq$ on a set X is a binary relation that is reflexive, antisymmetric, and transitive, i.e., it holds for all a, b and c in X that:

- $a \leq a$ (reflexivity)
- if $a \leq b$ and $b \leq a$ then $a = b$ (antisymmetry)
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (transitivity)

A partially ordered set of elements can be displayed as a directed acyclic graph (Figure 2.13). Such a graph does not have any cycles in it and contains edges which point in one direction. The order is here defined by the directed edges. The elements in the graph have only a partial order, because some of them cannot be related to each other. In Figure 2.13, the two nodes `N3` and `N4` are not related, because it cannot be determined, which one has a higher precedence in the order.

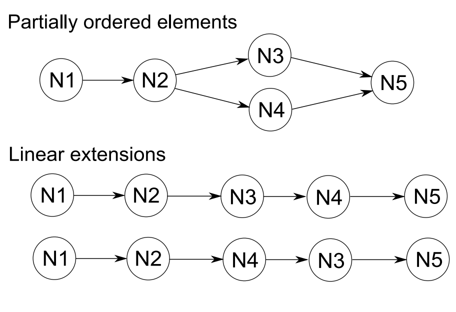


Figure 2.13: Topological sorting

Topological sorting is a method to sort a partially ordered set. It produces at least one linear extension of the partial order. In this extension, the elements are totally ordered. In the example in Figure 2.13, there exist two linear extensions of the original set. Since the nodes `N3` and `N4` do not relate to each other, there are two possibilities to sort them.

There exist several topological sort algorithms for finding all linear extensions (two efficient ones are described in [40] and [44]). For this thesis, a simple algorithm specified by Kahn [29], which only returns one possible linear extension, was extended in the way that it computes all the linear extensions of a partial order set (Listing 2.1). It is sufficient for simpler test cases. However, for more complex test cases, one of the efficient algorithms might be necessary.

```
LS := List containing all nodes with no incoming edges
LE := Empty list that will contain the nodes of one linear extension
L := Empty list that will contain all linear extensions
return getSorts(LS, LE, L)

function getSorts(list LS, list LE, list[list] L) returns list[list]
    for each node n in LS do
        LSlocal := LS
        LElocal := LE
        remove n from LSlocal
        insert n into LElocal
        if LElocal is a valid linear extension then
            insert LElocal into L
        else
            for each node m with an edge e from n to m do
                remove edge e from the graph
                if m has no other incoming edges then
                    insert m into LSlocal
            getSorts(LSlocal, LElocal, L)
    return L
```

Listing 2.1: Algorithm to retrieve all linear extensions from a partial order

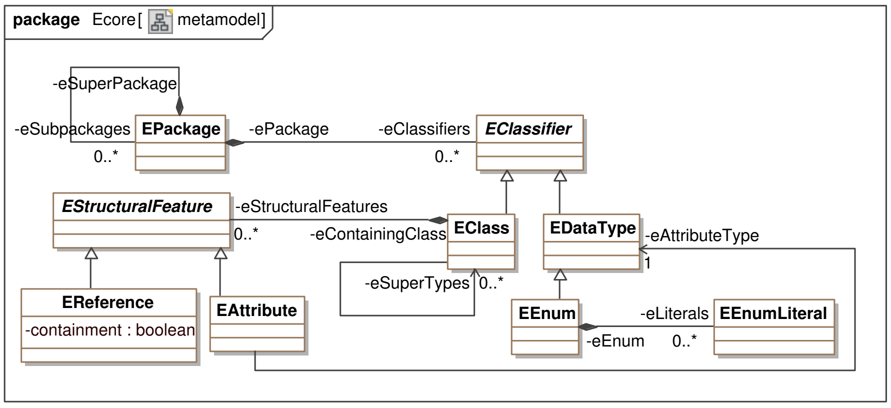


Figure 2.14: Part of the Ecore metamodel

The algorithm in Listing 2.1 calculates all linear extensions with the help of iteration and recursion. At the beginning, there is a number of start nodes, i.e., those that do not have any incoming edges. Each of those nodes is processed and moved to a sorted list. At the same time, all the children of the current node are determined and the corresponding edges are removed from the graph. A child that does not have any incoming edges after that, is shifted to the start nodes. The whole procedure is repeated recursively, until all the elements are moved through the list of the start nodes and then to the sorted list. Each recursion path has its own copies of those lists, and so each sorted list, which represents a linear extension, is added to a global list of lists that at the end is returned as result.

2.7 Eclipse Modeling Framework (EMF)

EMF is a “framework and code generation facility for building Java applications based on simple model definitions” [42]. It provides a modeling environment for Java programmers. Once a model of a system is created, Java classes and interfaces can be generated automatically that represent the whole model in terms of Java code. EMF contains a metamodel, called Ecore metamodel, that is used to build Ecore models. A model can either be created by hand from scratch or imported from one of several supported formats. For example, it is possible to convert an XML schema into an equivalent Ecore model. Other sources are UML models and annotated Java interfaces.

Conceptually, the Ecore metamodel can be compared to the part of the UML metamodel that is used to define class diagrams. That part of the UML is also reused by the MOF metametamodel, so the Ecore metamodel can also be used as a metametamodel (in the M3-level). Ecore is compatible to Essential MOF (EMOF), which is a subset of MOF.

An excerpt of the Ecore metamodel is shown in Figure 2.14. According to the Ecore notation, all Ecore element names are prefixed with an `E` to distinguish them from the corresponding UML elements. The container for the other Ecore elements is an `EPackage` element. Packages can contain other packages and classifiers. The element `EClassifier` is an abstract metaclass, hence a concrete element must be either a class or a data type. Classes can have other classes as supertypes. A class can contain structural features, i.e., references and attributes, as well as operations (not displayed here). A reference always points to another class (also not shown here). The `containment` attribute of the `EReference` element specifies the relation between the referenced class and the parent class. If the value of the attribute is `true`, then the referenced class is placed directly inside the parent class. Otherwise, the referenced class is located somewhere else in the model. The `EAttribute` element is another structural feature of a class. An attribute is always contained directly in its parent class. The type of an attribute is defined by the `EDataType` element. It can be either a simple type, for example `boolean` or `string`, or an enumeration, which consists of enumeration literals.

2.8 openArchitectureWare (oAW)

OpenArchitectureWare (oAW) [12] is a generator framework for model-driven software development. It is implemented in Java as a set of plug-ins for the Eclipse platform [5]. It supports model to model, model to text, as well as text to model transformations. A variety of models are supported, e.g., EMF, UML2, XML models and simple JavaBeans. It is also possible to integrate other model kinds. The models can be checked against constraints. Code for arbitrary programming languages can be generated from the models.

Expressions language

OpenArchitectureWare provides domain-specific languages for model to model transformations and code generation which are discussed in the subsequent sections. The languages share a common base language called the expressions language. The expressions language allows the other specialized languages to operate on the same models and metamodels. The following features are offered by the expressions language:

- Common arithmetic and boolean operators for numerical data types and strings, similar to Java operators;
- Conditional 'question mark' expression;
- Switch expression for choosing one execution branch out of several;
- Several expressions can be processed with the chain expression denoted by '->';

- Local variables can be defined with the `let` expression;
- Data types from the same heritage hierarchy can be casted exactly like in Java (static casting).

The oAW collection types `List` and `Set` can be processed by collection operations, which are similar to OCL operations.

- A subset of a collection for which a condition is true can be selected with the `select` statement.
- The `reject` statement removes the elements for which the condition evaluates to true.
- `typeSelect` selects all elements of the given type.
- A given attribute from all elements of a collection can be retrieved with the `collect` statement. A list of the attributes is returned. A shorthand for `collect` is the dot operator. For example, `employees.birthDate` returns a list with all the birth dates from all elements in the collection `employees`.
- The elements in a collection can be sorted with the statement `sortBy`.
- `forAll` is a boolean statement and returns true, if a given condition is true for all elements in the collection.
- If a condition evaluates to true for at least one element, then `exists` returns true, otherwise false.

Xtend

Xtend is a language for defining libraries for the other textual languages in oAW, but can also be used to specify model to model transformations and in-place model modifications. The main structure element of the language is *extension*. The syntax of an extension is as follows:

```
[create] ReturnType extensionName(ParamType1 paramName1, ...):
    expression-using-params;
```

The body of the extension consists of an expression defined in the oAW expressions language. The passed parameter values can be used to compute the return value. Instead of an expression, the extension body may contain a mapping to a Java method, in which case the specified method will be called. If the optional `create` statement is given, an object of the return type is automatically created and can be accessed in the expression through a `this` reference (or without any prefix as a shortcut).

An extension can be called in two ways: like a function or with a syntax that looks like a method call on an object. The following examples show the two possibilities.

```
extensionName(paramValue1, ...);
```

or

```
paramValue1.extensionName(...);
```

For a single value of a simple type as the first argument, the two versions have the same semantics. The value before the dot is just mapped to the first parameter of the extension. This makes extension calls like `5.add(1)` possible, which is the same as `add(5, 1)`. For an object, the meaning of the two versions could be different, if namely the object contains a method with the same name. In this case, the method has a precedence before the extension.

The biggest difference between the two possibilities of invoking an extension can be seen if the first argument is a collection, i.e., a list or a set. Xtend does not provide any structures for explicit loops. However, it is possible to invoke an *implicit* loop using the method call syntax. If an extension with a certain parameter type exists, and the extension is called with a *collection* containing objects or values of that type, then the second version will loop implicitly over all the items in the collection and invoke the extension once for each of them.

There is another implicit behavior involved in calling extensions which are prefixed with the `create` keyword. If the same create extension is called several times with the same set of parameter values, then the object created in the first call is returned every time. This behavior is useful in model to model transformations for creating references in the output model. Instead of searching explicitly in the model for the object to be referenced, simply the extension that created the object can be called one more time with the same set of arguments.

Xpand

The Xpand language is used in model to text transformations to control generation of code from models. The central concept in Xpand is *template*, specified as a `DEFINE` block. The following example displays the general structure of a template.

```
<<DEFINE templateName(formalParameterList) FOR MetaClassName>>
    <<FILE fileName.java->>
        sequence-of-statements
    <<ENDFILE>>
<<ENDDEFINE>>
```

A template is always defined for a certain metaclass from the metamodel of the used input model. The template in the above example contains a FILE statement which redirects all the output to the specified file. All the text in the body of the FILE statement including whitespaces is considered to be output. A minus sign before a closing guillemet bracket can be used to remove the directly following newline character from the output. Other templates, which do not contain a FILE statement, can be called, or *expanded*, in the body of the template. The output of those templates is also written to the file.

Another way to produce output is to compute it with statements defined in the oAW expressions language. These must be placed inside a DEFINE or FILE statement and enclosed in guillemets. Any strings that are returned from the expressions are also redirected to the generated text file.

Xpand defines some other structures to process the input model, which are explained below.

- The FOREACH statement loops over all elements of a collection.
- IF uses the expressions language for boolean guards to make conditional execution of statements possible.
- Local variables can be defined with the LET statement.

Workflows

The script that performs the model to model and model to text transformations using definitions in the languages described above, is called *workflow* and is executed by a *workflow engine*. A workflow is a sequential execution of special objects called *workflow components*. Each component fulfills a certain task in the transformation process. oAW contains several predefined workflow components for reading and writing files, executing the Xtend and Xpand definitions, checking constraints on a model, and other tasks.

The order and configuration of the workflow components is specified in an XML file, the workflow file, which can be read by the workflow engine. Here, the classes of the workflow components are referenced and the configuration arguments are given. The workflow engine instantiates objects using this information and executes them.

2.9 Testing And Test Control Notation Version 3 (TTCN-3)

TTCN-3 [16] is a language which was designed explicitly for specifying tests and it is standardized by the European Telecommunications Standards Institute (ETSI) [7]. While the previous versions were limited to testing telecommunication protocols, the current version can be used to specify tests for a number of application areas, including grid computing, real-time testing, and software testing in general. Both the asynchronous message-based communication and the synchronous procedure-based communication paradigms are supported.

TTCN-3 is an abstract language with different presentation formats. Apart from the textual format, there is a tabular and a graphical MSC-like format. The syntax of the textual format is similar to that of the programming languages C and C++. However, TTCN-3 is highly adapted to testing purposes. It contains special constructs — like test components, ports, templates, matching operators for matching test data, or a very fine-grained type system — which cannot be found in other programming languages.

Test definitions in TTCN-3 are organized in modules. The modules contain specifications for test cases, test components, the interface to the SUT, and data definitions. Apart from data types, constants, and variables, TTCN-3 offers port types, verdict types, timers, and templates. A port type defines ports which are used by the test components to send messages to the SUT and to each other, or to call operations in the SUT and in each other. Messages are represented as templates which can contain values or matching expressions. Verdicts are used to define the results of single test cases and of the entire tests. Timers make it possible to perform real-time tests and detect lost messages. Test cases are executed from a central point, the `control` part of the module.

2.10 Related Work

There are several approaches that are similar to the method described in this thesis. They all produce executable test cases from test models using model to model and/or model to text transformations, i.e., they follow the MBT or the MDT methodology as defined in Section 2.5. In the following, some of those approaches are discussed and the similarities as well as the differences to this method are pointed out.

The SAMSTAG method [24] makes it possible to generate test cases from a test architecture and system specification defined in Specification and Description Language (SDL) [28] and from behavior of test cases modeled in Message Sequence Charts (MSCs) [27]. The test architecture contains the test components and the System Under Test (SUT) as well as the communication interfaces between them. An MSC consists of lifelines corresponding to the objects of the test architecture which send messages with data units to each other. The output of the SAMSTAG tool, which is the implementation of the SAMSTAG method, are test cases in the testing language TTCN. The SAMSTAG method is similar to the method of this thesis. The major difference is that it does not have an intermediate model, but transforms the source model directly into code. Another difference is that the inner behavior of the SUT is considered, whereas in this thesis, the SUT is viewed as a black box.

The approach of Dai [21] is different with respect to the source of the test definitions. Test cases are not modeled independently from the system specification, but are directly derived from the system specification which is given in a UML model. In the first step, the UML model is analyzed and a UTP model is produced that contains the test cases. This transformation was implemented in Tefkat [14]. In the second step, the UTP model is mapped to TTCN-3. Instead of restricting UTP, extensions to the TTCN-3 language are proposed so that a one-to-one transformation would become possible.

There are several tools that follow the MBT principle, which means that they derive test cases from system specification models. In Elvior MOTES [6], system models are defined as finite state machines. Like in this thesis, test cases are generated in TTCN-3. The tool Conformiq [3] accepts system models specified in Java or UML. Test cases can be generated for different languages, for example, TTCN-3, Visual Basic, XML, or Python. TGV [15] uses a model defined as a Labeled Transition System (LTS) and generates TTCN test cases.

Like the approach in this thesis, the method of Zander et al. [45] is based on the MDA and MDT methodologies. UTP is also used to model tests and TTCN-3 is the target language for test cases. However, that method does not restrict UTP and therefore the model to text transformation is defined by general rules that produce TTCN-3 code which must be completed manually to become executable. The transformation was also implemented for the Eclipse platform, but in Java instead of languages that are designed specifically for model to model and model to text transformations.

The Eclipse plug-in oAW-Test [10] can be used to model own test cases and transform them automatically into test scripts for the tool JMeter [2]. Instead of UML or UTP, the models must be defined in a special XML format specified for the plug-in. The similarities here are that openArchitectureWare is used and that test case generation from test models is performed.

3 Test Case Generation using Model to Model Transformations

Like UML, UTP is not an exact programming language and can be used in different ways by different developers. The examples in the UTP specification [33] make this clear: They do not prescribe rules for the modeler to use UTP, but rather provide suggestions. For example, a default might be modeled as a state machine or as an interaction. Similarly, test cases can be modeled as interactions or as activities. Due to this fact, it is difficult to derive executable test cases from a model that is created without following certain rules. For this reason, UTP models that are supposed to be suitable for transformations, must be more restrictive than the examples in the specification. In Section 3.1, a variant of UTP called SWEUTP is presented, which is used in this thesis to model test cases. It is described in terms of general rules for the structure of models that it describes. The differences to the UTP standard are addressed. Additionally, formally defined restrictions for the models conforming to SWEUTP are presented in Section 3.2.

The goal of this thesis is to create a method to produce executable test cases from models. However, SWEUTP being a UML profile is a very complex modeling language. Hence, it is difficult to produce test cases directly from an SWEUTP model. For this reason, an SWEUTP model is first transformed to a simpler model that is easier to manage. In Section 3.3, the metamodel called SWETest is presented, which defines the structure of such models. The transformation from SWEUTP to SWETest is described in Section 3.4.

An interaction in both SWEUTP and SWETest defines behavioral statements, such as sending a message, which are partially ordered. Since test cases are modeled as interactions, there is a number of test cases defined in each interaction. To obtain all the linear extensions, i.e., all the possible test cases from an interaction, the statements can be sorted topologically. This process requires another model to model transformation which is presented in Section 3.5. An SWETest model is transformed again into an SWETest model. For each test case in the source model, the resulting model contains a number of test cases with topologically sorted statements.

The last step is the transformation of an SWETest model into executable code. In this thesis, the testing language TTCN-3 has been chosen as the target language of the transformation. This process is described in detail in Section 3.6.

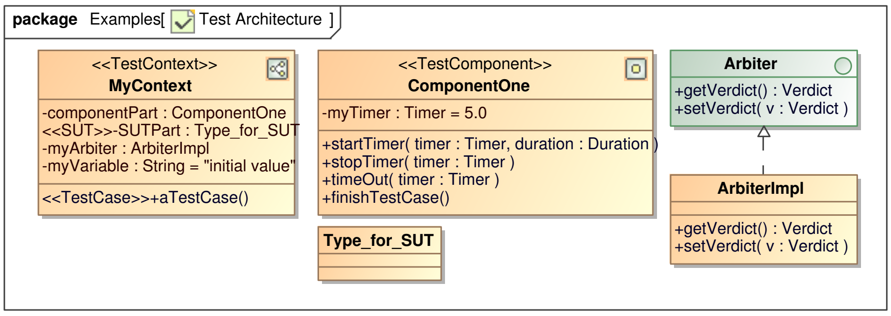

package Examples[Test Architecture]
<<TestContext>>
MyContext
-componentPart : ComponentOne
<<SUT>>-SUTPart : Type_for_SUT
-myArbiter : ArbiterImpl
-myVariable : String = "initial value"
<<TestCase>>+aTestCase()
<<TestComponent>>
ComponentOne
-myTimer : Timer = 5.0
+startTimer(timer : Timer, duration : Duration)
+stopTimer(timer : Timer)
+timeOut(timer : Timer)
+finishTestCase()
Type_for_SUT
Arbiter
+getVerdict() : Verdict
+setVerdict(v : Verdict)
ArbiterImpl
+getVerdict() : Verdict
+setVerdict(v : Verdict)

Figure 3.1: SWEUTP test architecture

3.1 SWEUTP

The UTP variation created for this thesis is called SWEUTP. It contains most of the elements from the UTP standard. Unlike the standard, SWEUTP has a defined structure that specifies, among other things, which elements are inside which other elements, or how exactly test cases are modeled. In this section, SWEUTP is described informally by explaining, what the models conforming to it must look like.

Besides attributes that represent test components, the SUT, and the arbiter, the test context can contain attributes of primitive or user-defined types (`myVariable` in Figure 3.1). Optionally, those attributes can have default values. Such attributes are considered global variables and can be used in the behaviors and parts of the test context, e.g., in the test cases or test components.

Similarly, test components can contain timer attributes which can be used in the test cases on the respective test components. Here, as well, the timers might have an initial value which is interpreted as the default duration of the timer. Additionally, a test component can contain operations that correspond to the actions in UTP.

The two attributes `SUTPart` and `myArbiter` of the test context (Figure 3.1) represent the SUT and the arbiter. More precisely, the SUT, like the test component, is a part in the composite structure of the test context. The two can be connected with each other via ports and connectors. The SUT has an empty class for its type. In general, the type of the SUT should be a class of, or an interface to the tested system, which is also modeled in UML and imported to the testing project. The arbiter type is represented by the `ArbiterImpl` class which realizes the UTP interface `Arbiter`. However, like for the timer attribute of the test component, it is sufficient to use the interface directly as the type for the arbiter in the test context. SWEUTP also contains a stereotype called `Arbiter` that can be applied to classes.

The scheduler interface from UTP is not used in SWEUTP. Instead, the starting of test cases is modeled in a test control. The test control is an activity, which is the classifier behavior of the test context. It controls the execution of test cases by referring to them through interaction use nodes. The verdict returned from a test case can be stored in an object node. This is modeled by an object node of type `Verdict` directly following the interaction use. The stored verdict might be used in subsequent conditions on edges which originate from decision nodes. In this way, it is possible to control the execution of test cases dependent on the outcome of the previous test cases. Since the UML standard does not define a syntax for boolean expressions, SWEUTP uses Java-like syntax for guards. An overview of the syntax is presented in Listing 3.1. The test case execution process can be logged using UTP log actions.

```
prog ::= expr+ | ELSE
expr ::= atom (AND | OR | GREATER | LESS | GREATEREQUALS | LESSEQUALS
          | EQUALS | NOTEQUALS ) atom
atom ::= (NOT)? (INT | ID | TRUE | FALSE |
          (LEFTPAREN expr RIGHTPAREN))
```

Listing 3.1: Syntax for SWEUTP guards

The referenced test cases in the test control are modeled as interactions. For every test case interaction, there exists a test case operation in the test context that refers to the respective interaction. Lifelines in the interaction represent the parts of the test context composite structure (test components and SUT) and the arbiter.

An interaction use can be placed on a lifeline. It refers to another interaction that will be executed in its place. The advantage of this is that one common behavior can be started in different test cases.

To stimulate the SUT, test component lifelines send signal messages to the SUT lifeline. The SUT can react by sending again signal messages to the test components. The messages are not sent to the SUT or components directly, but to ports contained in them, which are modeled in the composite structure of the test context. The sending and receiving ports are specified implicitly by referring to the connector which connects the two ports. A connector has to be referenced if the signal message is sent from a test component to the SUT. In the other direction it can be omitted. In this case, it means that the message can be received at any port in the test component. This can be used for unexpected messages coming from the SUT, for example to model erroneous situations. Similarly, a signal message sent from the SUT to a test component can, but does not have to, contain a signal. If no signal is contained in the message, it means that the test component is prepared to receive any possible signal. If a signal definition contains attributes, then the signal message may contain arguments of the same types. Those arguments can also be omitted. Another way to define such arguments are UTP wildcards. Instead of a concrete value, an arbitrary string can be passed, which is stereotyped as `LiteralAny` or `LiteralAnyOrNull`. For a better overview, the string itself should be, e.g., '?' or 'any' for `LiteralAny` and '*' or 'anyOrNull' for `LiteralAnyOrNull`. For an explicit omitting of a value, an argument can

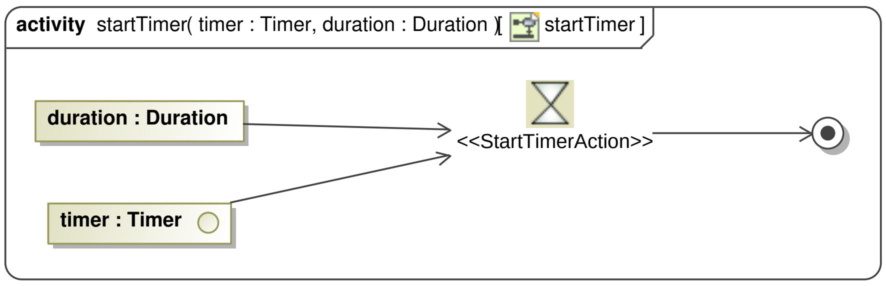

activity startTimer(timer : Timer, duration : Duration)[startTimer]
duration : Duration
timer : Timer
<<StartTimerAction>>

Figure 3.2: A UTP action in an activity

be a `LiteralNull` value from UML. The wildcards are useful for test components to receive messages with unknown or non-existing arguments.

Operation call messages can be sent to the arbiter. The components can call the operation `setVerdict` in the arbiter to indicate their verdicts. The operation has an argument of the enumeration type `Verdict`.

In SWEUTP, the UTP actions are used in activities. The test components can execute activities containing actions indirectly with a call message to self. The message to self indicates that a lifeline calls some behavior contained in the object which the lifeline refers to. The call message contains a reference to an operation in that object. The operation refers to an activity which contains the UTP action to be executed. As an example, the `StartTimerAction` within an activity is displayed in Figure 3.2. In this case, a timer reference and the duration until the timeout are passed to the action through activity parameter nodes. The operation that is called on the lifeline must have parameters of the same types. Other UTP actions which can be used in this way are:

- StopTimerAction, to which a timer reference must be passed;
- TimeoutAction, also with a timer reference;
- FinishAction, which does not need any parameters.

Defaults are useful to handle unexpected SUT behavior and error situations. Like test cases, a default is modeled as an interaction. A default interaction contains an `alt` block as its outer element. A default can be assigned to a lifeline in a test case by a comment attached to that lifeline with the word 'default' and the name of the default interaction.

The ports defined for the SUT and the test components in the test configuration have port types, which in SWEUTP are modeled as simple UML classes. Port types are not defined in the UTP standard. In SWEUTP, ports can only send and receive signal messages. The possible signals for those messages are connected by associations.

A signal may contain attributes of built-in or user-defined data types. User-defined types can be `DataType`, `PrimitiveType`, or `Enumeration` classifiers. Unstructured data types, i.e., with no attributes on themselves, have the meaning of renaming a built-in UML type. They refer to a UML type in their `baseClassifier` property to declare what type they redefine. This technique is useful to define own meaningful names for built-in types. Structured user-defined data types can model arbitrarily complex data structures. The values of those types, that can be sent in signal messages, are UML instances. The instances have slots, which contain values for each attribute of the structured data type. Recursively, a structured data type can contain attributes of other structured types. In this case, an instance slot contains a reference to another instance. If a value for an instance slot is not known or not important, it can be empty. As with arguments in signal messages, an instance slot may contain a `LiteralNull` to explicitly indicate an omitted value or it might contain an arbitrary string value, which is marked with one of the UTP wildcard stereotypes `LiteralAny` or `LiteralAnyOrNull`.

3.2 Constraining SWEUTP

The SWEUTP restrictions presented in this section are realized as OCL constraints. More precisely, the syntax corresponds to the OCL variant of the OCL Eclipse project [4]. This implementation contains some operations not available in the specification of OCL [35], for example, `getAppliedStereoypes()` and `returnResult()`, which slightly simplify the expressions.

The constraints presented here prescribe the structure of models that are conformant to SWEUTP. The context is always a UML metaclass, which means that the constraints are applied to elements of the M2-level. The context is not given specifically in a `context` definition, because all the constraints start with the name of the respective metaclass.

The test context is the container for the test configuration and the test cases. That is why every SWEUTP model has at least one test context. The following constraint ensures this. It is applied to all the instances of the metaclass `Class` that are contained in the model, i.e., all the classes in class diagrams. Only the classes that have the stereotype `TestContext` applied to them are selected and the constraint ensures that this collection contains at least one element.

```
Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext'))->notEmpty()
```

An SWEUTP model can contain more than one test context. In that case, each of the test cases is in its own package. This is stated by the next two constraints. The first constraint selects all the test cases and returns true, if every one of them is owned by a package. In contrast to that, the owner of a class in UML can also be for example another class. The second constraint ensures that the owner of every test context is unique, i.e., there are no two packages with test contexts that are the same.

```
Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext')).owner->
    forAll(oclIsTypeOf(Package))

Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext'))->isUnique(owner)
```

Each of the test contexts in the model has a classifier behavior. This behavior is an activity that contains instructions for executing the test cases, i.e., it models the test control. The following two constraints are used to ensure this. The first constraint checks for each test context, if its `classifierBehavior` property is not empty. Since it must be an activity, the second constraint makes sure that the classifier behavior is of type `Activity`.

```
Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext'))->forAll(classifierBehavior->
    notEmpty())

Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext'))->forAll(classifierBehavior.
    oclIsTypeOf(Activity))
```

The test configuration is given in the composite structure of the test context. It contains parts, ports and connectors. The next constraint returns true, if at least one part is contained in every test context. A part is an attribute that has its property `isComposite` set to true.

```
Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext'))->forAll(ownedAttribute->
    exists(isComposite))
```

The composite structure of the test context in an SWEUTP model always contains parts, which are either test components or an SUT. This constraint selects all test context attributes for which `isComposite` is true, and examines them. A part is an SUT, if the `SUT` stereotype is applied to it. In contrast to that, a test component part is not stereotyped directly. The stereotype `TestComponent` must be applied to its type, i.e., the classifier that represents the test component.

```
Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext')).ownedAttribute->
    select(isComposite)->forAll(getAppliedStereotypes()->
    exists(name='SUT') or type.getAppliedStereotypes()->
    exists(name='TestComponent'))
```

In SWEUTP, the messages between the test components and the SUT are exchanged through ports. This means, that the connectors in the test configuration always connect ports, and never directly the parts. This restriction is stated by the following constraint. Each end of every connector in the model contains the property `role`, which must be of type `Port`.

```
Connector.allInstances().end.role->forAll(oclIsTypeOf(Port))
```

Each port has a type, which is represented by a classifier. The next constraint ensures this by checking for every port in the model, if its `type` property is set.

```
Port.allInstances()->forAll(type->notEmpty())
```

Since test cases only make sense, if there is an SUT to be tested, the test configuretion of each test context contains a part representing the SUT. This constraint analyzes every test context and checks, if it contains a part among its attributes, which also has the `SUT` stereotype applied to it.

```
Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext'))->forAll(ownedAttribute->
    select(isComposite)->exists(getAppliedStereotypes()->
    exists(name='SUT')))
```

The SUT in a test context is always a property. All properties must have a type, so the next constraint makes sure that each SUT contains a reference to its type. The constraint selects all attributes of all test contexts, filtrates all SUTs from the resulting collection, and checks, if the `type` property of each SUT is not empty.

```
Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext')).ownedAttribute->
    select(getAppliedStereotypes()->exists(name='SUT'))->
    forAll(type->notEmpty())
```

The main behavior in SWEUTP is given by test cases. A model conforming to SWEUTP contains at least one test case. Test cases are represented by operations in the test context, which are stereotyped by `TestCase`. This next constraint analyzes all operations of every test context, and checks, if at least one operation exists that contains the stereotype `TestCase`.

```
Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext'))->forAll(ownedOperation.
    getAppliedStereotypes()->exists(name='TestCase'))
```

The test case operations do not contain the behavior for the test cases. They only point to the right behavior, which is located elsewhere in the model. In SWEUTP, the behavior type for test cases is interaction. The constraint below selects all the test case operations of all test contexts and ensures for each operation, that it includes the property `method`, which points to the behavior, as well as that the referenced behavior is of type `Interaction`.

```
Class.allInstances()->select(getAppliedStereotypes()->
    exists(name='TestContext')).ownedOperation->
    select(getAppliedStereotypes()->exists(name='TestCase'))->
    forAll(method->notEmpty()
    and method->oclIsTypeOf(Interaction))
```

SWEUTP supports the three interaction operators `alt`, `opt`, and `loop`. The next constraint selects all instances of the `CompinedFragment` metaclass and checks for each of them, if the `interactionOperator` property is set either to the enumeration literal `alt`, the enumeration literal `opt`, or the enumeration literal `loop`.

```
CombinedFragment.allInstances()->
    forAll(interactionOperator=InteractionOperatorKind::alt
    or interactionOperator=InteractionOperatorKind::opt
    or interactionOperator=InteractionOperatorKind::loop)
```

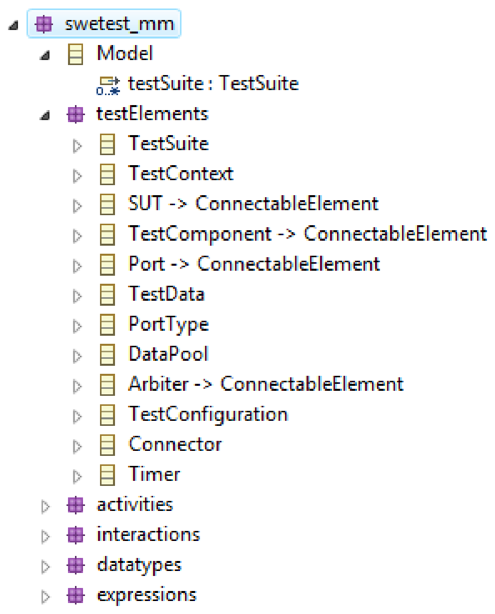
All the messages used by the test components to stimulate the SUT, as well as the answer messages from the SUT, have names. The messages used for communication in SWEUTP are asynchronous messages containing signals. The following constraint analyzes all messages and returns true, if all the messages of the sort `asynchSignal` have a name which does not equal to an empty string.

```
Message.allInstances()->
    select(messageSort=MessageSort::asynchSignal)->
    forAll(name<>'')
```

3.3 The SWETest Metamodel

SWETest is a metamodel designed specifically for representing tests. It belongs to the M2-level and is comparable to the UML metamodel with UTP. However, it is a much simpler metamodel, containing only metaclasses for testing purposes. SWETest conforms to the Ecore metamodel, hence Ecore is in this context a metametamodel. Since SWETest is a metamodel, the Ecore EClass element is called *metaclass* in the following. All the other Ecore elements are also not referred to by the specific Ecore names. For example, an EPackage is just called package and an EAttribute is called attribute. In this section, the SWETest metamodel is mostly described from the perspective of a possible model that conforms to SWETest. For example, 'a test case' means here 'an instance of the `TestCase` metaclass'. So, if no metaclass name is mentioned specifically, a model from the M1-level is being described.

All SWETest metaclasses are contained in the package `swetest_mm` (Figure 3.3). The metaclass `Model` defines the container for all the classes of models conforming to SWETest.



```
swetest_mm
  Model
    testSuite : TestSuite
  testElements
    TestSuite
    TestContext
    SUT -> ConnectableElement
    TestComponent -> ConnectableElement
    Port -> ConnectableElement
    TestData
    PortType
    DataPool
    Arbiter -> ConnectableElement
    TestConfiguration
    Connector
    Timer
  activities
  interactions
  datatypes
  expressions
```

Figure 3.3: Overview of the SWETest metamodel

The rest of SWETest is partitioned in five packages containing further metaclasses. The partitioning is only used for structuring purposes, i.e., the metaclasses from different packages can have relations to each other. The separate packages are described below.

- The package `testElements` contains metaclasses to model the static structure of tests. A test suite, for example, represents a consistent and independent testing unit with its own test context, test components, an arbiter, and test data definitions. The SUT is also represented by a metaclass. The test components and the SUT contain ports to communicate with each other.
- The `activities` package holds metaclasses that model behaviors containing actions. The actions are modeled as activity nodes connected by activity edges.
- Metaclasses for modeling the behavior of test cases and defaults are contained in the package `interactions`. Interactions consist of lifelines, which refer to test components, arbiters, and the SUT from the package `testElements`. The communication between those elements is modeled by messages sent between the lifelines.
- All the metaclasses that define data are contained in the `datatypes` package. This includes data for the messages as well as parameter and argument definitions for interactions and activities.

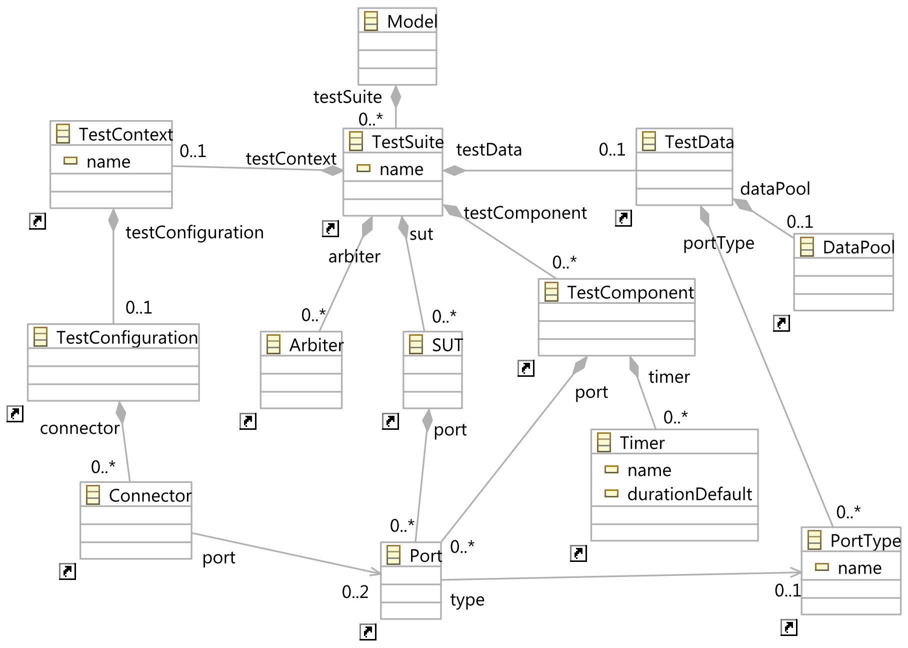


Figure 3.4: The `testElements` *package of SWETest*

- In the package `expressions`, metaclasses to represent conditions are contained. For every boolean expression operator like 'equals' or 'and' there is a corresponding metaclass that holds the operands which can be simple values or again operator metaclasses.

In Figure 3.4, all the metaclasses of the package `testElements` and their relations to each other are displayed. Additionally, all the relations to metaclasses from the other packages are shown in Figure 3.5. A model can contain an arbitrary number of test suites. A test suite can hold elements described in the following enumeration.

- The test context serves as a container for the test cases, modeled as interactions, and a test control, modeled as an activity (3.5). Other interactions and activities can be referred to with the corresponding references. Attributes and one test configuration can also be contained in the test context.

- The test data element holds all the data definitions. Signals and port types are included directly in the test data. All the data types and instances are contained in its child element data pool.

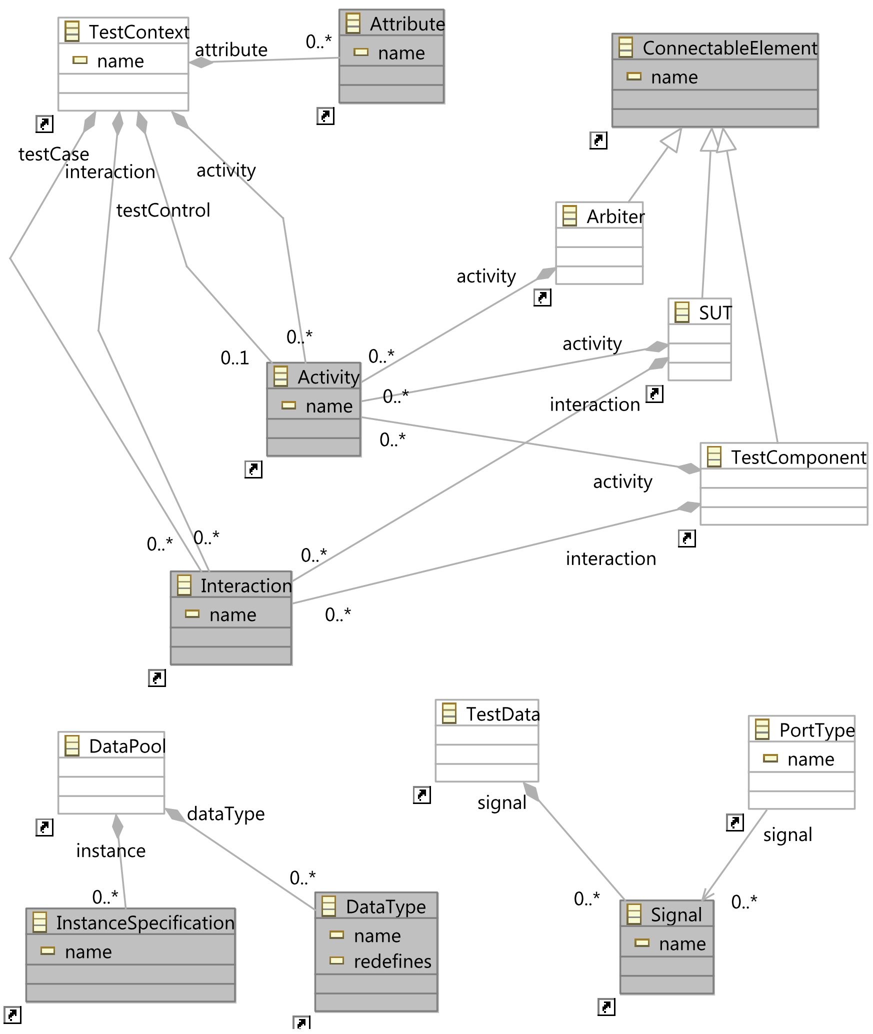


Figure 3.5: Relations of test elements to elements in other SWETest packages

- The three active test elements test component, arbiter, and SUT are descendants of the connectable element, which is used for references from lifelines. An arbiter can contain activities. A SUT and a test component can have activities, interactions, and ports. Additionally, test components can contain timers with duration defaults.

The test configuration from the test context consists of connectors. Each of the connectors has at most two references to ports. A port contains a reference to its type, which itself references a number of signals.

An activity can have parameters, activity edges, and activity nodes as its children (Figure 3.6). The activity edge and the activity node have references to each other. An activity node can refer to arbitrarily many incoming and outgoing edges. An activity edge, in contrast, can only have at most one node as its `source` reference and one node as its `target` reference. A `Condition` class can be contained inside an activity edge. The `ActivityNode` metaclass has descendants, which are described below.

- An activity parameter node is a means of storing a reference to a parameter of the activity.

- A call behavior action can have arguments and refers to an interaction.

- The other six actions represent behavior indicated by their names. A log action contains an attribute called `value`, which stores data to be logged.

- Decision and merge nodes are nodes for conditional execution.

- An object node represents an object, whose type is given by the referenced `DataType` metaclass.

The contents of the `interactions` package are displayed in the Figure 3.7. The parts of an interaction are parameters (see below), lifelines, and interaction fragments. An interaction fragment points to the lifelines it belongs to through its reference `covered`. A lifeline has a type, which is a connectable element. Moreover, a lifeline can contain a number of defaults, each of them referencing to an interaction, in which their behavior is specified. The metaclass `InteractioinFragment` is a superclass for other metaclasses, representing more specific concepts.

- An interaction use stands for a call of another interaction. The link is stored in the `interaction` reference. If the referenced interaction contains parameters, the interaction use can contain arguments, represented as value specifications.

- An interaction fragment can be a message. Since messages are sent from one lifeline to another, the `Message` metaclass contains the references `sourceLifeline` and `targetLifeline`. More specialized messages are call messages and signal messages.

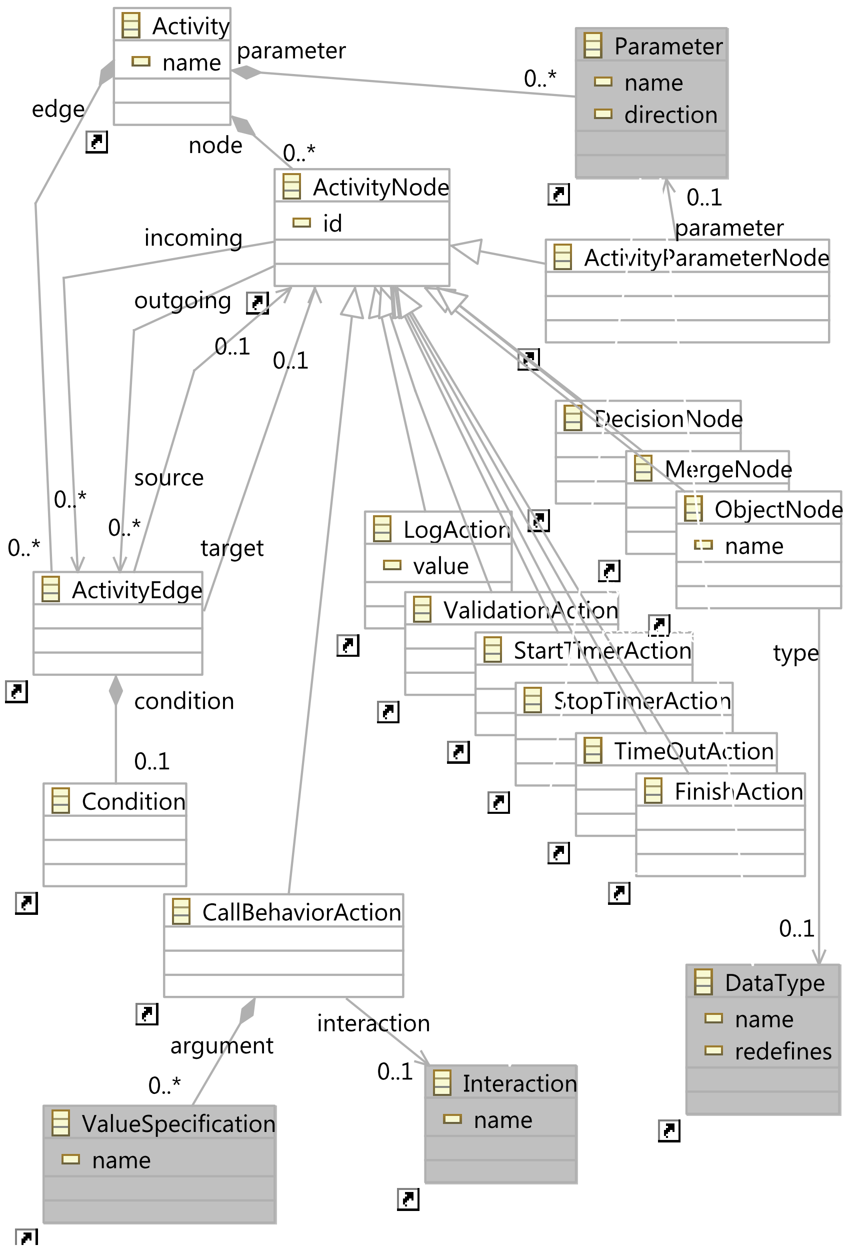


Figure 3.6: The `activities` *package of SWETest and relations to elements from other packages*

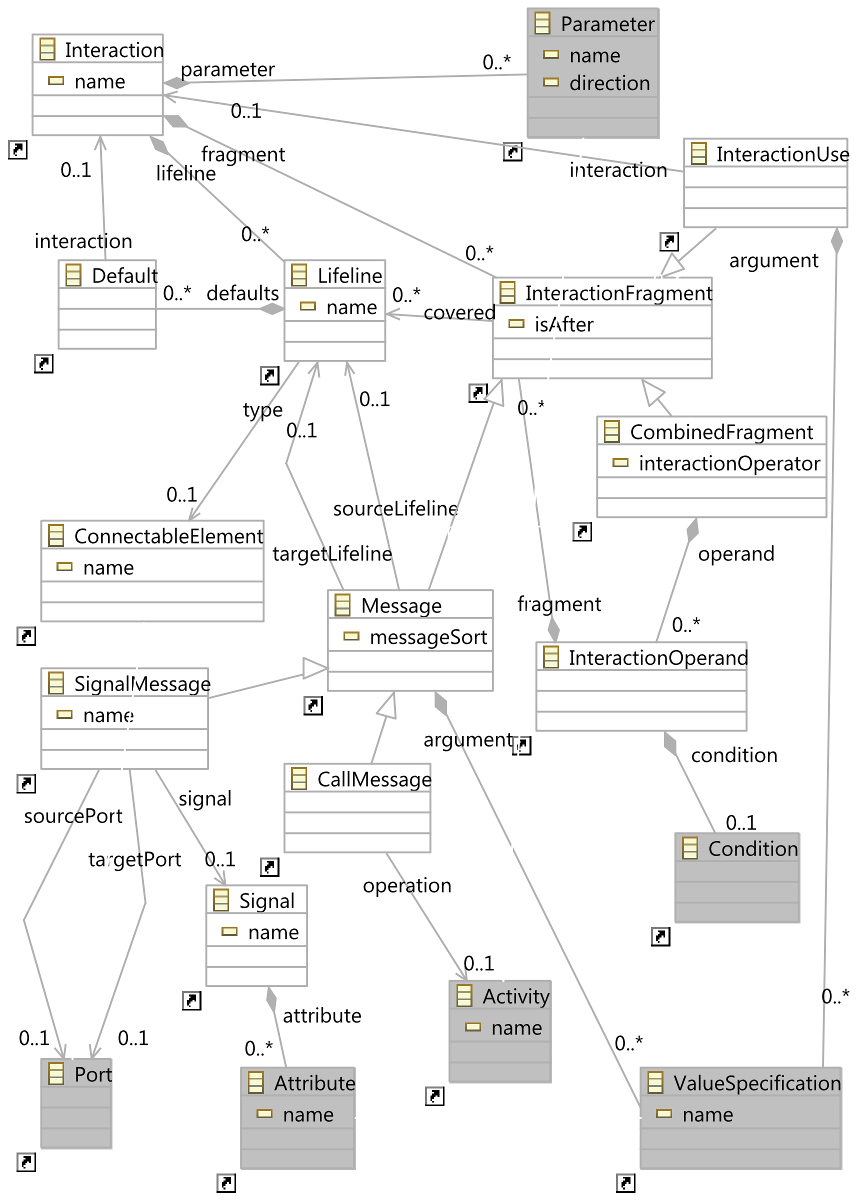


Figure 3.7: The `interactions` *package of SWETest and relations to elements from other packages*

A call message stores a reference to an activity to be called. A signal message refers to a signal, which can contain a number of attributes. Signal messages are sent from one port to another, which is why they have references to source and target ports.

- A combined fragment consists of interaction operands, which can contain a condition defining a guard for their execution. An interaction operand contains again interaction fragments.

The metaclasses in the package `datatypes` represent data concepts (Figure 3.8). The element `ValueSpecification` is the superclass for all metaclasses representing data *values*. A literal specification can be either a wildcard value or a value of a primitive type. The `value` attribute stores the actual value of the respective type. An instance of the `VariableName` metaclass holds a string value of a variable name. `NativeCode` may be used to store some platform-specific code. A timer reference is a link to a timer from the `testElements` package. A duration value contains a value specification to store either a literal float or a variable name.

An instance value has a reference to an instance specification, which stores a number of value specifications. The referenced data type defines the structure for its instance specifications by containing attributes, which again reference `DataType` instances for the definition of their types. An attribute can have default value specified as a value specification. Like the attributes, parameters also refer to a data type. A data type with no inner structure may have the `redefines` attribute set, which indicates what other type this one is having as a base.

The last possible value specification is an enumeration value. It stores an enumeration literal and references the enumeration containing that literal. `Enumeration` is a subclass of `DataType` and consists of enumeration literals. The value in the enumeration literals is stored as a string value.

The package `datatypes` additionally contains four enumerations, displayed in Figure 3.9. These enumerations can only define *static* values and should not be confused with the previously described metaclasses that store user-defined *dynamic* enumeration values. The enumeration `ParameterDirectionKind` is used in parameters to specify their direction in the activity or interaction. The `MessageSort` enumeration describes the kind of a message. A verdict can have the values defined in the `Verdict` enumeration. The last one, which is called `InteractionOperatorKind`, consists of literals specifying the kind of a combined fragment.

The `expresions` package holds metaclasses to define boolean conditions. The metaclass `Condition` contains a containment reference to `Expression`, which is a superclass for several more specific expression kinds. An expression can be itself a value specification. So, a 'not' expression can have as its `operand` attribute either a value, or recursively another expression. Similarly, all the binary expressions have two operands, which can be values or

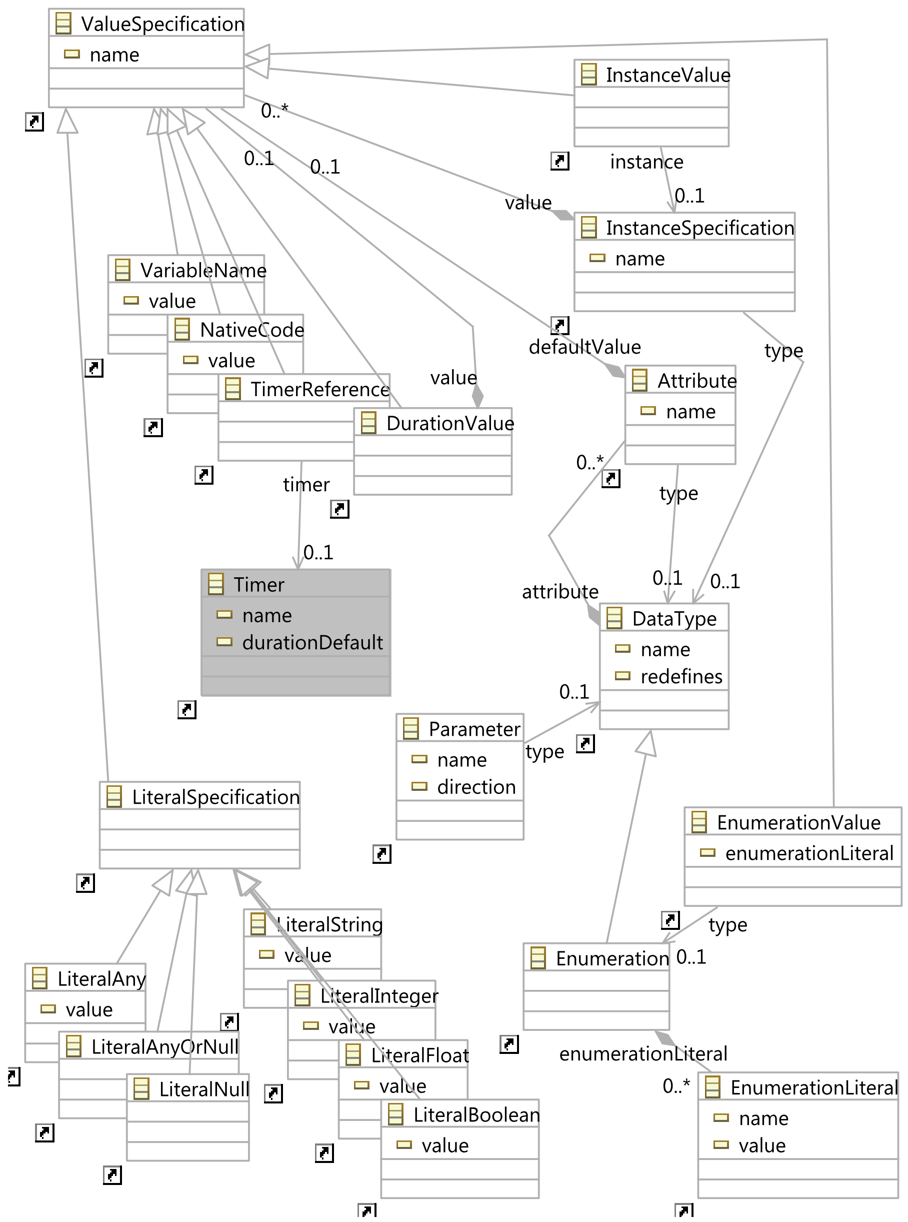


Figure 3.8: `datatypes` *package of SWETest and relations to elements from other packages*

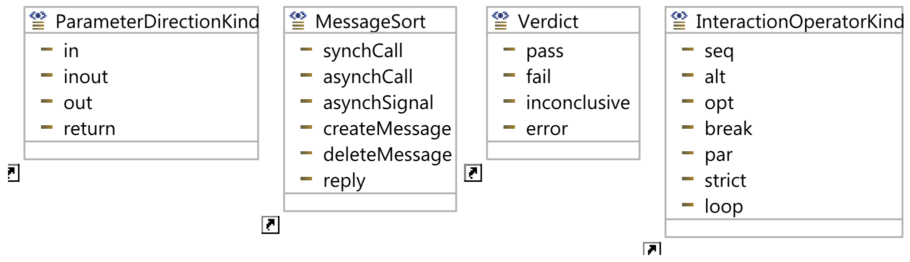


Figure 3.9: Enumerations from the `datatypes` *package*

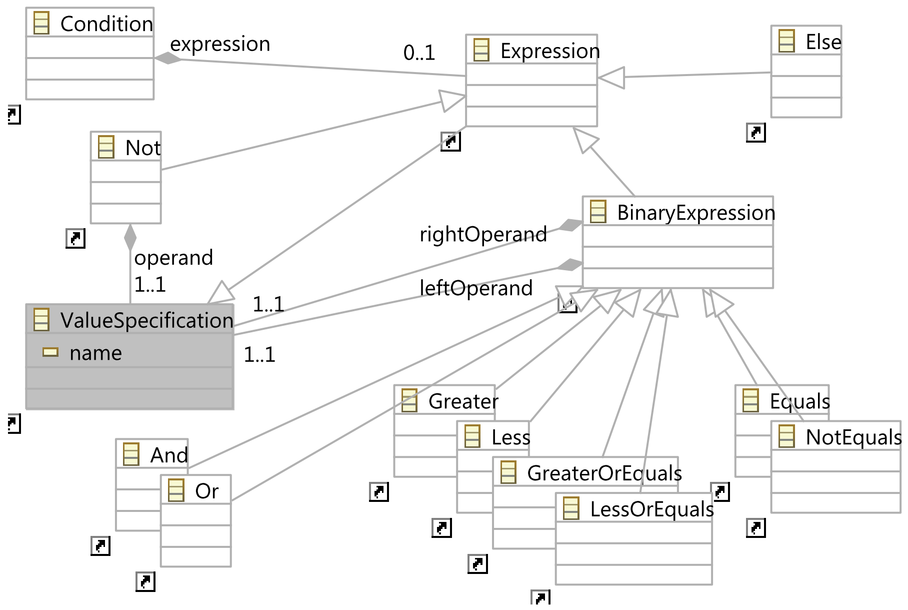


Figure 3.10: The `expressions` *package of SWETest*

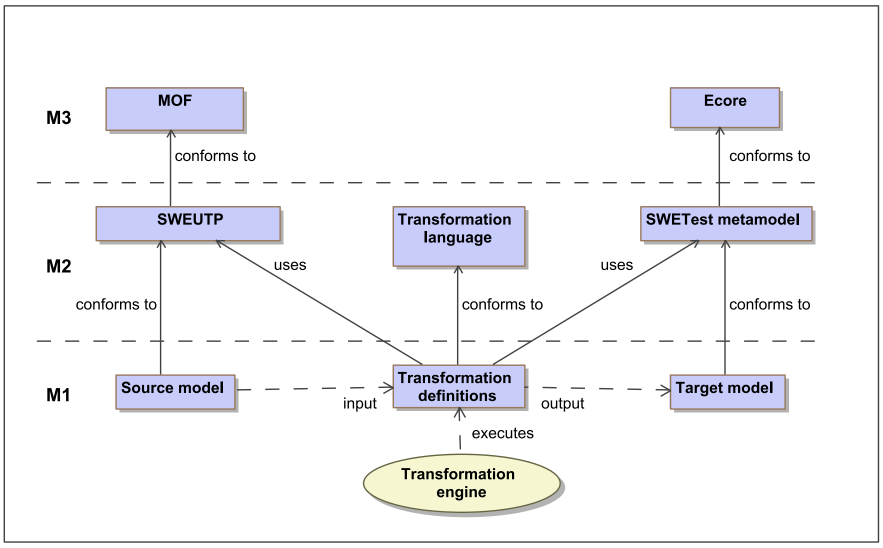


Figure 3.11: Transformation from SWEUTP to SWETest

expressions. The `Else` metaclass represents a special kind of expression used in combined fragments of type `alt`.

3.4 SWEUTP to SWETest Transformation

The first step of producing code for test cases from an SWEUTP model is to transform the model into a simplified, more manageable model. This intermediate model conforms to the SWETest metamodel specified in the previous section. In this section, the transformation of an SWEUTP model into an SWETest model is described. Figure 3.11 shows an overview of the transformation process. It can be compared to a model to model transformation in MDA. However, there are some differences to that method, which are listed in the following.

- The target model does not conform to a metamodel that conforms to MOF. The metametamodel used here is Ecore, which is similar to a subset of MOF called Essential MOF (EMOF).
- The transformation language is not a metamodel, but is defined as a textual notation. However, in the sense of MDE, the language can be viewed as a metamodel and the concrete transformation definitions as a model.

- MDA defines the transformation of a platform-independent model into a platform-specific model before generating code. In contrast to that, both SWEUTP models and SWETest models are platform-independent. SWETest can be viewed as a version of SWEUTP that is simplified and more structured.

The transformation process is described below in terms of mapping rules in tables. Each line in a table defines a mapping from left to right. There are three kinds of rules that can be contained in a table: rules for metaclasses, for metaattributes, and for metaclass references[1]. The tables show the types of the elements[2] involved in the transformation, i.e., the metaclasses or the metaattributes from the corresponding metamodels. However, the meaning is that each element of the given source type maps to one element of the given target type. The definitions for UML metaclasses (which are reused in SWEUTP) can be found in Appendix A and in in the UML specification [38].

The rules also define the *context* of the current element in the respective model. For SWEUTP metaclasses, the context is given informally, explaining where in the model the element of that type is placed. The formal definition of the context is not possible here, because the SWEUTP metamodel does not specify a strict structure for the placement of all the elements. Therefore, elements in an SWEUTP model could be scattered, only connected by references. In contrast to that, each SWETest element has a specifically defined parent element. The context of the SWETest metaclasses is given as a prefix to the metaclass name. The prefix consists of names of the container metaclasses that precede the current metaclass in the SWETest metamodel, separated by colons. In some cases however, the absolute context in SWETest cannot be given, because there are several possibilities or even an infinite number of possibilities due to recursive structures. In that situation, only a relative context is specified, starting with three dots. The context of the metaattributes and references is always specified as an OCL-like path from the current metaclass. The target metaclass of a reference is always given after an arrow. To achieve a better overview, the context of the target metaclass is omitted.

Main Test Containers

Test cases, test components, and all the other elements necessary to design a complete test are included in a container. In SWEUTP, that container is a package. In an SWETest model, all the test elements are contained in a test suite. Several of those packages or test suites can be bundled inside a model. First, the transformation rules for those containers are described.

[1]In the Ecore language, this kind of reference is a non-containment reference. The other kind of reference, containment reference, is defined as parent–child relations in the text between the tables

[2]The word *element* is used here both to refer to an instance of a metaclass and to an attribute of such an instance

SWEUTP element	*SWETest element*
`Model`	`Model`

An SWEUTP `Model` element that contains the entire model is transformed into a corresponding SWETest `Model` element, which also serves as a container for all other model elements. Since both elements are the top level containers in the respective models, and both models contain exactly one element of the respective type, the transformation is finished when the one is mapped to the other. All the other elements are either direct or indirect children of the two `Model` elements and are transformed in the process as defined by the rules in the following.

SWEUTP element	*SWETest element*
`Package` containing a class with stereotype «TestContext»	`Model:TestSuite`
`Package.name`	`TestSuite.name`

The source model contains one or more test contexts, which are all placed in separate packages. For each of those packages, there is a test suite in the target model. The name of the corresponding package is taken as the name for the test suite. An SWETest model always consists of one or more test suites, because the `TestSuite` elements are the only children of the SWETest `Model` element. A test suite is a self-contained unit, which has all the necessary information to test a system. That information is contained in the elements inside the the test suite, which are described in the following subsections.

Data Definition Elements

The elements discussed in this subsection define types for data values, ports, and signals. These elements are used as types for references by other SWETest elements. For that reason, transformations including the data definition elements are discussed here first so that the other transformations can be described later completely.

SWEUTP element	*SWETest element*
`Package` of the test context	`Model:TestSuite:TestData`

For each package with a test context in the SWEUTP model, there exists a `TestData` element in the SWETest model. More precisely, the data in that SWEUTP package is used by the descendants of the `TestData` element. The test data is a container for all elements storing data related information.

SWEUTP element	*SWETest element*
`Signal` referenced by a send signal event	`Model:TestSuite:TestData: Signal`
`Signal.name`	`Signal.name`

SWEUTP signals map to SWETest signals. Since SWEUTP `Signal` elements can be placed anywhere in the model, only signals that are actually used are selected. This means that only signals sent between the test components and the SUT become `Signal` elements in SWETest and are placed as children into the test data. The name of each signal in the source model is the same as the new name in the target model.

SWEUTP element	*SWETest element*
`Class` referenced as type for a port in the SUT or a test component	`Model:TestSuite:TestData: PortType`
`Class.name`	`PortType.name`
`Class.attribute.type` → `Signal`	`PortType.signal` → `Signal`

In SWEUTP, each port has a type modeled as a class. For each of such classes that are actually referenced from a port, there exists a `PortType` element in the SWETest model. The corresponding name of the class is taken for the name of the port type. A port type in SWEUTP can contain a number of references to signals, which are types of its attributes. Those references map to `signal` references in SWETest that point to corresponding `Signal` elements.

SWEUTP element	*SWETest element*
`Package` of the test context	`Model:TestSuite:TestData: DataPool`

For each package containing a test context in SWEUTP, there is a `DataPool` element in SWETest. The package does not map to the corresponding data pool, but the data in the package is used for transformations involving the children of the data pool. The data pool is a container for all instances and data types that are not used to define ports or signals, but rather to define *values* to be sent in messages.

SWEUTP element	*SWETest element*
`Classifier` referenced in an instance specification, attribute, or parameter as type	`Model:TestSuite:TestData: DataPool:DataType`
`Classifier.name`	`DataType.name`
`Classifier.baseClassifier`	`DataType.redefines`

The package of the test context (including the test context itself) can contain instance specifications, attributes, and parameters, which have type references. Each classifier defined as type in one of those elements is mapped to a `DataType` element in SWETest. The value of the `name` attribute is mapped correspondingly. If the `baseClassifier` attribute in the classifier is set, then it is taken as the value for the `redefines` attribute.

SWEUTP element	*SWETest element*
`Enumeration` referenced in an interaction	`Model:TestSuite:TestData: DataPool:Enumeration`
`Enumeration.name`	`Enumeration.name`

Each enumeration that is used in a test case or another interaction in the SWEUTP model maps to an enumeration in the SWETest model. Like data types, enumerations in SWETest are children of a data pool. The name of the source element is the same as the name of the target element.

SWEUTP element	*SWETest element*
`EnumerationLiteral` inside an enumeration	`Model:TestSuite:TestData: DataPool:Enumeration: EnumerationLiteral`
`EnumerationLiteral.name`	`EnumerationLiteral.name`
`EnumerationLiteral.specification. body`	`EnumerationLiteral.value`

Like the enumerations in SWEUTP, the enumerations in SWETest consist of enumeration literals. Besides the name of the literal, a specification can be present in SWEUTP. This is an opaque expression whose `body` property specifies the constant that the literal represents in the enumeration. This property becomes a `value` attribute in the SWETest enumeration literal.

SWEUTP element	*SWETest element*
`InstanceSpecification` referenced from an instance value in a message as argument	`Model:TestSuite:TestData: DataPool: InstanceSpecification`
`InstanceSpecification.name`	`InstanceSpecification.name`
`InstanceSpecification.classifier` $\longrightarrow$ `Classifier`	`InstanceSpecification.type` $\longrightarrow$ `DataType`

The last kind of children of the data pool in SWETest are instance specifications. For each instance specification in SWEUTP that is actually sent in a message as argument, there is an instance specification in SWETest with the same name. The reference to a classifier in SWEUTP becomes a `type` reference to a `DataType` element in SWETest.

Test Structure Elements

The main static elements of a test are the test context and active test elements like test components, arbiters, and the SUT. These elements define the structure of a test specification. In this subsection, the transformations involving such elements are defined.

SWEUTP element	*SWETest element*
`Property` of the test context with stereotype «SUT»	`Model:TestSuite:SUT`
`Property.type.name`	`SUT.name`

Each test context in an SWEUTP model contains a stereotyped property that represents the SUT. This property becomes a `SUT` element in the SWETest model. The SUT property in SWEUTP has a type, which is a class and is referenced by the `type` attribute. The name of that class is the same as the name of the SUT in the SWETest model.

SWEUTP element	*SWETest element*
`Class` with stereotype «TestComponent», referenced as type for a part in the test context	`Model:TestSuite: TestComponent`
`Class.name`	`TestComponent.name`

In SWEUTP, test components are specified as classes with `TestComponent` stereotypes. Those classes actually serve as types for test component instances in the test context, which participate in test cases. Such an instance is a property of the test context, or, more precisely, a part in the composite structure of the test context. Each class serving as a type for such a property maps to a `TestComponent` element in the SWETest model, including the `name` attribute.

SWEUTP element	*SWETest element*
`Port` inside a SUT type	`Model:TestSuite:SUT:Port`
`Port` inside a test component	`Model:TestSuite:TestComponent:Port`
`Port.name`	`Port.name`
`Port.type` → `Class`	`Port.type` → `PortType`

In SWEUTP, there are ports inside test components and the SUT, which are used to transfer messages with signals. Each of those ports maps to a port in SWETest and is a child of the corresponding test component or SUT. The respective name of the port is mapped to a `name` attribute. The type of the port, which is a class in SWEUTP, becomes a reference to a `PortType` element.

SWEUTP element	*SWETest element*
`Property` of a test component with the type 'Timer'	`Model:TestSuite:` `TestComponent:Timer`
`Property.name`	`Timer.name`
`Property.defaultValue.body`	`Timer.durationDefault`

Test components can contain properties that represent timers. For each of those properties, there is a `Timer` element in SWETest with the same name as the property. If there exists an opaque expression in the attribute `defaultValue`, the body of the expression is mapped to a `durationDefault` attribute in the SWETest timer.

SWEUTP element	*SWETest element*
`Class` with stereotype «Arbiter», referenced as type for an attribute in the test context	`Model:TestSuite:Arbiter`
`Class.name`	`Arbiter.name`

Similar to the test components above, there are classes in the SWEUTP model that represent arbiter types, i.e., those are used as types for arbiter instances in the test context. Each of those classes corresponds to an `Arbiter` element in the test suite of the SWETest model.

SWEUTP element	*SWETest element*
`Property` of the test context, type name is 'Arbiter'	`Model:TestSuite:Arbiter`
`Property.type.name`	`Arbiter.name`

As an alternative to using a class for an arbiter type in SWEUTP, an interface named `Arbiter` can serve as type for a test context property. In this case, the property itself maps to an `Arbiter` element in SWETest. The `name` attribute of the property's type is taken as name for the arbiter in SWETest.

SWEUTP element	*SWETest element*
`Class` with stereotype «TestContext»	`Model:TestSuite:TestContext`
`Class.name`	`TestContext.name`

Each class stereotyped as `TestContext` in SWEUTP becomes a respective `TestContext` element in SWETest. The name of the class corresponds to the name of the test context in SWETest. The test context in SWETest contains children representing the test configuration, test cases, the test control, and other behavioral features. Transformations involving those children are described next.

SWEUTP element	*SWETest element*
Composite structure of `Class` with stereotype «TestContext»	`Model:TestSuite:TestContext:TestConfiguration`

Each test context in SWEUTP contains a composite structure which represents the test configuration. Each of those composite structures corresponds to a `TestConfiguration` element in SWETest.

SWEUTP element	*SWETest element*
`Connector` inside test context	`Model:TestSuite:TestContext:TestConfiguration:Connector`
`Connector.end.role` → `Port`	`Connector.port` → `Port`

A composite structure of a test context contains ports connected by connectors. For each of the connectors there is a corresponding `Connector` element in SWETest inside the test configuration. The ports in SWEUTP are referenced by the `role` attribute of connector ends. Each of those roles maps to a port reference in SWETest. The referenced port can be either inside a SUT or inside a test component.

SWEUTP element	*SWETest element*
`Property` inside test context	`Model:TestSuite:TestContext:Attribute`
`Property` inside signal	`Model:TestSuite:TestData:Signal:Attribute`
`Property` inside class or data type	`Model:TestSuite:TestData:DataPool:DataType:Attribute`
`Property.name`	`Attribute.name`
`Property.type` → `Type`	`Attribute.type` → `DataType`

Properties are structural features of test contexts, signals, classes, and data types. A property from SWEUTP is always mapped to an `Attribute` element inside the according element in SWETest. The name of the property is taken as the name for the attribute. The reference to the type of the property becomes a reference to a corresponding `DataType` element in SWETest.

Interactions

Interactions can be contained in several elements in both models. An interaction is used to define a test case, a general function, or the implementation of a default. The transformation rules for interactions and their structure elements are defined next.

SWEUTP element	*SWETest element*
`Interaction` inside test context	`Model:TestSuite:TestContext: Interaction`
`Interaction` inside test context with stereotype «TestCase»	`Model:TestSuite:TestContext: Interaction` as `testCase`
`Interaction` inside test component	`Model:TestSuite: TestComponent:Interaction`
`Interaction` inside SUT	`Model:TestSuite:SUT: Interaction`
`Interaction.specification.name` or `Interaction.name`	`Interaction.name`

An interaction inside a test context, a test component, or a SUT is transformed into an interaction inside a corresponding element in SWETest. A special case of interaction is a test case. In SWEUTP, a test case can only occur inside a test context. Such an interaction is transformed to an `Interaction` element and is placed inside the test context in SWEUTP as a child named `testCase`. An interaction in SWEUTP can have a `specification` attribute set, i.e., there is an operation pointing to this interaction. In this case, the name of the operation is preferred and taken as the name for the interaction in SWETest. Otherwise, the name of the source interaction is mapped to the name of the target interaction.

SWEUTP element	*SWETest element*
`Lifeline` inside interaction	`...:Interaction:Lifeline`
`Lifeline.represents` $\rightarrow$ `ConnectableElement`	`Lifeline.type` $\rightarrow$ `ConnectableElement`

Lifelines contained in interactions also map to lifelines in SWETest. The reference called `represents` that points to a connectable element becomes a `type` reference in SWETest, which also points to an according connectable element. This is a generalized rule for three other elements, since the connectable element is in SWETest a superclass for SUT, test component, and arbiter.

SWEUTP element	*SWETest element*
`Comment` in an interaction annotating a lifeline and starting with 'default'	`...:Interaction:Lifeline: Default`
Substring of `Comment.body` $\rightarrow$ `Interaction`	`Default.defaultInteraction` $\rightarrow$ `Interaction`

The use of a default in SWEUTP is represented by a comment that starts with the string 'default' and is attached to a lifeline. For each of those comments, there exists a `Default` element in SWETest as child of a lifeline. The substring of the `body` attribute following 'default' is a name of an interaction specifying the behavior of the default. The reference `defaultInteraction` refers to the interaction in SWETest that has that name.

The elements specifying the behavior of an interaction are interaction fragments. In both models, the element named `InteractionFragment` is a supertype for other, more specialized elements. In the following, the transformations of the subtypes of the interaction fragment are defined.

SWEUTP element	*SWETest element*
`InteractionUse` in an interaction	`...:Interaction: InteractionUse` as `fragment`
`InteractionUse.refersTo` → `Interaction`	`InteractionUse.interaction` → `Interaction`
`InteractionUse.covered` → `Lifeline`	`InteractionUse.covered` → `Lifeline`

An interaction use is an interaction fragment that specifies the call of another interaction. The element in SWEUTP is transformed to a corresponding element in SWETest and placed in an interaction. The `refersTo` attribute in SWEUTP holds a reference to the called interaction and is mapped to an `interaction` attribute in SWETest. The references to the covered lifelines in SWEUTP are the according references in SWETest to the mapped lifelines.

SWEUTP element	*SWETest element*
`MessageOccurrenceSpecification` that is received by a lifeline, with a send signal event	`...:Interaction:SendMessage` as `fragment`
`MessageOccurrenceSpecification.message.name`	`SendMessage.name`
`MessageOccurrenceSpecification.message.messageSort`	`SendMessage.messageSort`
`MessageOccurrenceSpecification.event.signal` → `Signal`	`SendMessage.signal` → `Signal`
`MessageOccurrenceSpecification.message.sendEvent.covered` → `Lifeline`	`SendMessage.sourceLifeline` → `Lifeline`
`MessageOccurrenceSpecification.covered` → `Lifeline`	`SendMessage.targetLifeline` → `Lifeline`
The two last ones above	`SendMessage.covered` → `Lifeline`
`MessageOccurrenceSpecification.message.connector.end.role` → `Port`	`SendMessage.sourcePort` → `Port`
`MessageOccurrenceSpecification.message.connector.end.role` → `Port`	`SendMessage.targetPort` → `Port`

In SWEUTP, there are two kinds of message occurrence specifications at the receiving end of a message: those that reference a send signal event and those that do not. The transformation for the first kind is defined in the table above. Such a message occurrence specification maps to a `SendMessage` element in SWETest. Since it is also an interaction fragment, the send message is placed in the interaction in the attribute named `fragment`. The two attributes `name` and `messageSort` of the `message` reference become attributes of the send message with the same names. The occurrence specification has a reference to an event. The reference to a signal in this event becomes a reference to the corresponding signal in the SWETest model.

All the references to the involved lifelines and ports are also transformed to SWETest references. In SWEUTP, references to the source lifelines are stored in message occurrence specifications that belong to send points of messages. The current message occurrence specification belongs to the receive end of the message, so its counterpart can be found through the `sendEvent` reference of the message. The `covered` association that points to the lifeline sending the message is mapped to the reference `sourceLifeline` in SWEtest. The `covered` association from the current occurrence specification becomes a `targetLifeline` reference. Additionally, both associations are mapped to a multivalued attribute named `covered` in SWETest. References to the ports through which the message is sent are mapped to corresponding references in SWETest. In SWEUTP, the ports are referenced in the connector ends as `role` associations. The connector end that belongs to the source lifeline holds the outgoing port. The reference to that port maps to a corresponding reference named `sourcePort`. Similarly, the reference to the receiving port in SWEUTP maps to a reference named `targetPort`.

SWEUTP element	*SWETest element*
`MessageOccurrenceSpecification` that is received by a lifeline, without a send signal event	`...:Interaction:CallMessage` as `fragment`
`MessageOccurrenceSpecification.message.messageSort`	`CallMessage.messageSort`
`MessageOccurrenceSpecification.event.operation.method` → `Activity`	`CallMessage.operation` → `Activity`
`MessageOccurrenceSpecification.message.sendEvent.covered` → `Lifeline`	`CallMessage.sourceLifeline` → `Lifeline`
`MessageOccurrenceSpecification.covered` → `Lifeline`	`CallMessage.targetLifeline` → `Lifeline`
The two last ones above	`CallMessage.covered` → `Lifeline`

If a message occurrence specification does not have a send signal event, it means that the associated message is a call message, which references an operation to be executed. That occurrence specification is transformed to a call message in SWETest. The value of the message sort from the associated SWEUTP message is copied to the corresponding `messageSort` attribute in the call message. In SWEUTP, the message occurrence specification contains a reference to an event that refers to an operation. The behavior of the operation is specified by an activity, which is referenced by the `method` attribute of the operation. That activity maps to an SWETest activity, which is linked from the call message by the attribute `operation`. The transformations for the source and target lifelines, as well as for the `covered` attribute, are the same as the ones for send messages described above.

SWEUTP element	*SWETest element*
`CombinedFragment` in an interaction	`...:Interaction: CombinedFragment` as `fragment`
`CombinedFragment.interactionOperator`	`CombinedFragment.interactionOperator`
`CombinedFragment.covered` → `Lifeline`	`CombinedFragment.covered` → `Lifeline`

The last kind of interaction fragments in SWEUTP is the combined fragment. A combined fragment is mapped to an SWETest element with the name `CombinedFragment` and, like the other fragments, put into the `fragment` attribute of the interaction. The interaction operator and the covered reference are transformed to the corresponding SWETest elements with the same names.

SWEUTP element	*SWETest element*
`InteractionOperand` inside a combined fragment	`...:Interaction: CombinedFragment: InteractionOperand` as `operand`

A combined fragment consists of interaction operands in both models. An interaction operand is a container for interaction fragments, i.e., it can contain messages, interaction uses, or other combined fragments. Like in an interaction, the fragments here are contained in a multivalued attribute named `fragment`. The transformation rules for the interaction fragments are defined above.

Activities

Like interactions, activities can be contained inside several possible elements in both models. An activity can represent supporting behavior for the test cases, e.g., execution of a timer, as well as the ordered execution of the test cases, i.e., the test control. In the following, the transformation rules for activities and their containing children are described.

SWEUTP element	*SWETest element*
`Activity` that is the classifier behavior inside a test context	`Model:TestSuite:TestContext:` `Activity` as `testControl`
`Activity.name`	`Activity.name`

The classifier behavior of a test context is an activity in SWEUTP. It is used as a control behavior for test cases that are also contained in the test context. That activity is transformed to an activity in SWETest with the same name. In SWETest, the activity is a child named `testControl` inside the test context.

SWEUTP element	*SWETest element*
`Operation` inside a test component	`Model:TestSuite:` `TestComponent:Activity`
`Operation` inside an arbiter	`Model:TestSuite:Arbiter: Activity`
`Operation.name`	`Activity.name`

SWEUTP operations in test components and arbiters also map to activities in SWETest. The parent and the name of the operation are the same as the parent and the name of the corresponding activity. If the operation contains a reference to an activity, then the the inner structure of the SWEUTP activity is transformed to corresponding elements that are children of the SWETest activity. An activity mainly consists of activity nodes and activity edges. Transformation rules for those elements are discussed next.

SWEUTP element	*SWETest element*
`ActivityNode` inside activity	`...:Activity:ActivityNode`
`ActivityNode.incoming` $\rightarrow$ `ActivityEdge`	`ActivityNode.incoming` $\rightarrow$ `ActivityEdge`
`ActivityNode.outgoing` $\rightarrow$ `ActivityEdge`	`ActivityNode.outgoing` $\rightarrow$ `ActivityEdge`

An activity node is in SWETest again an activity node. This rule is a general rule for all subtypes of the `ActivityNode` element, both in SWEUTP and in SWETest. Each activity node contains references to incoming edges and references to outgoing edges. The references are mapped to corresponding references in SWETest. In the following, the rules involving concrete activity nodes are defined.

SWEUTP element	*SWETest element*
`SendObjectAction` with stereotype 'LogAction' inside activity	`...:Activity:LogAction`
`SendObjectAction.target.value` or `SendObjectAction.request.value`	`LogAction.value`

Each send object action in SWEUTP that is stereotyped as log action becomes a `LogAction` element in SWETest. A send object action might have the value of either the `target` or the `request` attribute set. That value maps to a `value` attribute of the log action in SWETest.

SWEUTP element	*SWETest element*
`CallBehaviorAction` inside activity	`...:Activity: CallBehaviorAction`
`CallBehaviorAction.behavior` → `Interaction`	`CallBehaviorAction. interaction` → `Interaction`

A call behavior action also maps to a call behavior action. In SWEUTP, a call behavior action has a reference called `behavior` that points to the interaction that should be executed. This reference corresponds to the `interaction` reference in SWETest.

SWEUTP element	*SWETest element*
`CallOperationAction` referring to operation with stereotype 'TestCase' inside activity	`...:Activity: CallBehaviorAction`
`CallOperationAction.operation.method` → `Interaction`	`CallBehaviorAction. interaction` → `Interaction`

A call operation action in SWEUTP might refer indirectly to an interaction that implements a test case by containing a reference to a test case operation. In this case, the call operation action is transformed to a call behavior action like in the previous rule. Here, the `method` reference of the contained operation becomes the reference to the called interaction in SWETest.

SWEUTP element	*SWETest element*
`CentralBufferNode` inside activity	`...:Activity:ObjectNode`
`CentralBufferNode.name`	`ObjectNode.name`
`CentralBufferNode.type` → `Classifier`	`ObjectNode.type` → `DataType`

Central buffer nodes are used to store a value or an object of a certain type. For each central buffer node in SWEUTP, there is an object node in SWETest with the same name. The reference to the type of the buffer node, which is a classifier, becomes a reference to a data type in SWETest.

SWEUTP element	*SWETest element*
`ActivityParameterNode` inside activity	`...:Activity: ActivityParameterNode`
`ActivityParameterNode.parameter` → `Parameter`	`ActivityParameterNode. parameter` → `Parameter`

An activity parameter node refers to a parameter of the enclosing activity. The node itself, as well as the reference to a parameter, are transformed to corresponding elements in SWETest.

SWEUTP element	*SWETest element*
`CallOperationAction` with stereotype 'StartTimerAction' inside activity	`...:Activity: StartTimerAction`
`CallOperationAction` with stereotype 'StopTimerAction' inside activity	`...:Activity: StopTimerAction`
`AcceptEventAction` with stereotype 'TimeOutAction' inside activity	`...:Activity: TimeOutAction`
`CallOperationAction` with stereotype 'ValidationAction' inside activity	`...:Activity: ValidationAction`
`OpaqueAction` with stereotype 'FinishAction' inside activity	`...:Activity:FinishAction`
`DecisionNode` inside activity	`...:Activity:DecisionNode`
`MergeNode` inside activity	`...:Activity:MergeNode`

The table above shows the transformation rules for the remaining possible activity nodes. Those nodes do not contain any attributes or references, so the rules are combined in one table. The different kinds of stereotyped actions are transformed to actions in SWETest with the names of the corresponding stereotypes. Decision nodes and merge nodes are mapped to a respective `DecisionNode` and `MergeNode` elements in SWETest.

SWEUTP element	*SWETest element*
`ActivityEdge` inside activity	`...:Activity:ActivityEdge`
`ActivityEdge.source` → `ActivityNode`	`ActivityEdge.source` → `ActivityNode`
`ActivityEdge.target` → `ActivityNode`	`ActivityEdge.target` → `ActivityNode`

Activity edges represent directed connections between the activity nodes. An activity edge is in SWETest also an activity edge. The references `source` and `target` also map to corresponding references in SWETest.

SWEUTP element	*SWETest element*
`Parameter` inside activity	`...:Activity:Parameter`
`Parameter` inside operation	`...:Activity:Parameter`
`Parameter` inside interaction	`...:Interaction:Parameter`
`Parameter.name`	`Parameter.name`
`Parameter.direction`	`Parameter.direction`
`Parameter.type` → `Type`	`Parameter.type` → `DataType`

Activities, operations, and interactions in SWEUTP can contain parameters. The parameters from activities and operations become parameters of activities in SWETest. Interaction parameters are also children of interactions in SWETest. The two attributes `name` and `direction` map to corresponding attributes in SWETest. The reference to the type of the parameter becomes a reference to a `DataType` element.

Value Specifications

Value specifications are used in places where concrete values are needed. For example, they can represent simple data values like strings or integers, or they can point to complex values, i.e., instances and enumerations. The transformation rules presented in this subsection involve `ValueSpecification` elements, as well as elements of inherited types.

SWEUTP element	*SWETest element*
`ValueSpecification` in a value specification action of an interaction use, referenced as `argument`	`...:Interaction: InteractionUse: ValueSpecification` as `argument`
`ValueSpecification` inside a message, referenced as `argument`	`...:Interaction:Message: ValueSpecification` as `argument`
`OpaqueExpression` inside a call message as `argument`, type of the operation parameter is 'Duration'	`...:DurationValue: ValueSpecification` as `value`
`ValueSpecification` inside a slot of an instance specification, referenced as `value`	`Model:TestSuite:TestData: DataPool: InstanceSpecification: ValueSpecification` as `value`
`ValueSpecification` inside an attribute, referenced as `defaultValue`	`...:Attribute: ValueSpecification` as `defaultValue`
`ValueSpecification.name`	`ValueSpecification.name`

A value specification can be contained inside several elements in SWEUTP. An interaction use can have arguments for the called interaction. In SWEUTP, those arguments are represented by value specification actions containing value specifications. The value specifications are mapped to arguments in SWETest, which are represented by `ValueSpecification` elements stored as `argument` children in an interaction use. The same transformation is applied to value specifications inside messages. An opaque expression inside a call message is treated as a special case if the the corresponding parameter of the referenced operation is of type `Duration`. In this case, the argument is transformed into a `DurationValue` element containing a value specification. The context in the target model is cannot be specified in the table above, because the duration value is itself a value specification and thus there are a number of possible contexts.

Instance specifications have slots, which contain value specifications. Those are transformed into value specifications directly inside the corresponding instance specifications in SWETest. Default values in attributes also become value specifications in SWETest referenced as `defaultValue`. A value specification can have a name, which is also transformed into a name in SWETest. However, the the name is only set if the parent is a slot in an instance specification, because in all the other cases it is not required.

In the following, transformation rules for the subtypes of `ValueSpecification` are defined. Since the context is the same, i.e., there is a number of possibilities, it is not given in the following rules.

SWEUTP element	*SWETest element*
`LiteralInteger`	`...:LiteralInteger`
`LiteralInteger.value`	`LiteralInteger.value`

SWEUTP element	*SWETest element*
`LiteralBoolean`	`...:LiteralBoolean`
`LiteralBoolean.value`	`LiteralBoolean.value`

SWEUTP element	*SWETest element*
`LiteralString`	`...:LiteralString`
`LiteralString.value`	`LiteralString.value`

The three elements `LiteralInteger`, `LiteralBoolean`, and `LiteralString` map to corresponding elements in SWETest. The `value` attribute is just copied to the SWETest version of the attribute for each element.

SWEUTP element	*SWETest element*
`LiteralString` whose value can be converted to a floating point number	`...:LiteralFloat`
`LiteralString.value`	`LiteralFloat.value`

SWEUTP does not contain elements for storing floating point numbers. However, if a literal string contains a value that can be parsed and transformed into a float, that string literal becomes a `FloatLiteral` element in SWETest. The string value becomes a floating point value inside the literal float.

SWEUTP element	*SWETest element*
`ValueSpecification` with stereotype «LiteralAny»	`...:LiteralAny`
`ValueSpecification` with stereotype «LiteralAnyOrNull»	`...:LiteralAnyOrNull`
`LiteralNull`	`...:LiteralNull`

SWEUTP contains three elements to specify wildcard values for matching purposes. A value specification stereotyped as `LiteralAny` or `LiteralAnyOrNull` is transformed into a `LiteralAny` and a `LiteralAnyOrNull` SWETest element, respectively. A literal null is just mapped to an element also called `LiteralNull`.

SWEUTP element	*SWETest element*
`InstanceValue`	`...:InstanceValue`
`InstanceValue.instance` $\rightarrow$ `InstanceSpecification`	`InstanceValue.instance` $\rightarrow$ `InstanceSpecification`

An instance value is a value specification that holds a reference to an instance specification elsewhere in the model. Instance values map again to instance values in SWETest. The `instance` reference points in SWETest to the according instance specification transformed before.

SWEUTP element	*SWETest element*
`InstanceValue` whose `instance` reference points to an enumeration literal	`...:EnumerationValue`
`InstanceValue.instance.name`	`EnumeraitonValue.enumerationLiteral`
`InstanceValue.instance.owner` $\rightarrow$ `Enumeration`	`EnumeraitonValue.type` $\rightarrow$ `Enumeration`

In SWEUTP, there is no special element for values from enumerations. Instead, an instance value is used as a container for a reference to an enumeration literal. Such an instance value is transformed into an `EnumerationValue` element. The name of the referenced literal becomes an attribute that is called `enumerationLiteral` in SWETest. The reference to the owner of the literal, which is an enumeration, maps to a `type` reference, which also points to an enumeration.

Some values that are useful to model test cases do not have corresponding elements in SWETest. Such values are represented as opaque expressions. The expressions have different meanings in different contexts. The following rules describe transformations involving opaque expressions.

SWEUTP element	*SWETest element*
`OpaqueExpression` as argument in a call message, the corresponding operation parameter is of type 'Timer'	`...:TimerReference`
`OpaqueExpression.body` → `Timer`	`TimerReference.timer` → `Timer`

An opaque expression in SWEUTP can contain a name of a timer. Such an expression can be contained in a call message that calls an operation. The expression is an argument for that operation. The corresponding parameter definition of the operation must be of type `Timer`. In that case, the opaque expression is transformed into a `TimerReference` element. The `body` attribute of the expression contains the name of the referenced timer. Out of this information, a reference called `timer` is produced in SWETest.

SWEUTP element	*SWETest element*
`OpaqueExpression` as argument in a call message, the corresponding operation parameter is of type 'Duration'	`...:DurationValue`

An opaque expression is also used in SWEUTP to specify a duration for a timer. Here, the parameter type of the operation is `Duration`. In this case, the opaque expression is transformed into a `DurationValue` element in SWETest. The duration value contains a child element of type `ValueSpecification`, which is described at the beginning of this subsection.

SWEUTP element	*SWETest element*
`OpaqueExpression` as argument in a call message, the corresponding operation parameter is also of type 'OpaqueExpression'	`...:NativeCode`
`OpaqueExpression.body`	`NativeCode.value`

For exceptional cases, it is possible to input a line of code for the target testing language directly into an SWEUTP model. In this case, the parameter type of the called operation must be `OpaqueExpression`. Such an expression is mapped to a `NativeCode` element and the body of the expression is copied to the `value` attribute.

SWEUTP element	*SWETest element*
`OpaqueExpression` as argument inside a signal message	`...:VariableName`
`OpaqueExpression.body`	`VariableName.value`

A test context can attributes with default values, that are treated like global variables in the test cases. In that situation, it is possible to send such a variable in a signal message by giving the name of the attribute inside an opaque expression, which is transformed into a `VariableName` element with the `value` attribute containing the variable name itself.

Boolean Expressions

Boolean expressions are stored inside guards in SWEUTP. A guard can occur in a combined fragment of an interaction or in an edge of an activity. Transformations for guards are defined next.

SWEUTP element	*SWETest element*
`OpaqueExpression` inside activity edge, referenced as `guard`	`...:Activity:ActivityEdge: Condition`
`LiteralString` inside interaction operand, referenced as `guard.specification`	`...:Interaction: CombinedFragment: InteractionOperand: Condition`

SWEUTP does not define a syntax for boolean expressions, so those are just stored inside opaque expressions and literal strings. If an activity edge contains a guard, it is transformed into a condition in SWEUTP. Interaction operands inside combined fragments can contain guards that have a literal string with a boolean expression. Such a literal string is also transformed into a `Condition` element.

SWEUTP element	*SWETest element*
`body` of the `OpaqueExpression` from above	`...:Condition:Expression`
`value` of the `LiteralString` from above	`...:Condition:Expression`

The two attributes `body` and `value` from the elements of the previously described transformations both map to expressions in SWETest, which are contained inside a condition. `Expression` is a supertype for other concrete expressions, which are described below.

SWEUTP element	*SWETest element*
`body` or `value` containing the operator '`!`' and one operand	`...:Condition:Not`

SWEUTP element	*SWETest element*
Single operand in `body` or `value`	`...:Condition:Not: ValueSpecification` as `operand`

If the body or value from above contain the 'not' operator displayed as an exclamation mark and an operand, each of them maps to a `Not` expression in SWETest. The operand, which in SWEUTP is represented by a substring of body or value, maps corresponds to a value specification inside of the `Not` element. In SWETest, `Expression` inherits from

`ValueSpecification`, so the operand can either be an arbitrary value from the previous subsection, or recursively another expression.

SWEUTP element	*SWETest element*
`body` or `value` containing two operands and a binary operator	`...:Condition: BinaryExpression`

SWEUTP element	*SWETest element*
Left operand in `body` or `value`	`...:Condition: BinaryExpression: ValueSpecification` as `leftOperand`
Right operand in `body` or `value`	`...:Condition: BinaryExpression: ValueSpecification` as `rightOperand`

`BinaryExpression` is a supertype for other possible expressions that are described below. Similar to the rules of the unary 'not' expression, the string value from SWEUTP is transformed into a `BinaryExpression` element. The two operands are again mapped to value specifications.

SWEUTP element	*SWETest element*
`body` or `value` containing two operands associated with '`&&`'	`...:Condition:And`
`body` or `value` containing two operands associated with '`||`'	`...:Condition:Or`
`body` or `value` containing two operands associated with '<'	`...:Condition:Less`
`body` or `value` containing two operands associated with '<='	`...:Condition:LessOrEquals`
`body` or `value` containing two operands associated with '>'	`...:Condition:Greater`
`body` or `value` containing two operands associated with '>='	`...:Condition:GreaterOrEquals`
`body` or `value` containing two operands associated with '=='	`...:Condition:Equals`
`body` or `value` containing two operands associated with '`!=`'	`...:Condition:NotEquals`

The table above lists the rules for all possible binary expressions. If the operator in the SWEUTP string value is the one defined on the left, then the string is transformed into the corresponding element on the right.

SWEUTP element	*SWETest element*
`body` or `value` containing the word '`else`'	`...:Condition:Else`

The last expression variant in SWEUTP contains only the word `else`. It is used for choosing an interaction operand or an activity edge if all the other expressions are false. The word is mapped to an empty element called `Else`.

3.5 SWETest to SWETest Transformation

An SWETest model produced by the transformation rules in the last subsection contains test cases whose interaction fragments are partially ordered. If one test case contains at least two test components, then the order of their messages to the SUT cannot be determined. Likewise, other interaction fragments that involve only some of the test components cannot be ordered in a sequence. This means that the test case has a non-deterministic behavior.

To obtain deterministic behavior of a test case, its fragments must be sorted topologically. This is done with the help of the extended version of Kahn's algorithm from Chapter 2, which computes all linear extensions of the partial order set defined in the interaction. The partial order is specified by the rule, that only fragments on the same test component lifeline are totally ordered. Each resulting linear extension maps to one separate new test case. In addition to the sorting, all the test components are merged into one test component that contains all the ports, timers, and behaviors.

In this section, the rules for transforming an SWETest model into another SWETest model are defined. The target model contains a number of test cases with topologically sorted fragments and only one merged test component. Most of the elements from the source model are not changed in the target model, so the rules below define only parts that are different in the target model. Attributes and references inside metaclasses that stay the same, are also not shown in the transformation tables.

Source element	*Target element*
All `Model:TestSuite: TestComponent` elements	One `Model:TestSuite:TestComponent` element
`TestComponent.name`	`TestComponent.name =` 'CompositeComponent'

In order for the target model to contain test cases with deterministic behaviors, there must be only one test component that stimulates the SUT. All the test components from the source model become one test component in the target model. The name of the new component is `CompositeComponent`.

Source element	*Target element*

`Model:TestSuite:`
`TestComponent:Timer`
	`Model:TestSuite:TestComponent:`
	`Timer` in composite component

`Timer.name`	`TestComponent.name` and `Timer.name`

Source element	*Target element*

`Model:TestSuite:`
`TestComponent:Port`
	`Model:TestSuite:TestComponent:Port`
	in composite component

`Port.name`	`TestComponent.name` and `Port.name`

Source element	*Target element*

`Model:TestSuite:`
`TestComponent:Activity`
	`Model:TestSuite:TestComponent:`
	`Activity` in composite component

`Activity.name`	`TestComponent.name` and `Activity.name`

All the contents of the test components in the source model is moved into the new composite component in the target model. This includes timers, ports, and activities. For each of those elements, the name in the target model is composed from the name of the original test component and own name. This prevents collisions if elements of the same type in different original test components have the same name.

Source element	*Target element*

`Model:TestSuite:TestContext:`
`Connector`
	`Model:TestSuite:TestContext:`
	`Connector`

`Connector.port` → `Port` in all test components	`Connector.port` → `Port` in composite component

Connectors have references to ports that can be contained inside of test components. In that case, such a reference is mapped to a reference that points to the corresponding port in the composite component.

Source element	*Target element*

`Model:TestSuite:TestContext:`
`Interaction` as `testCase`
	Several `Model:TestSuite:TestContext:`
	`Interaction` elements as `testCase`

`Interaction.name`	`Interaction.name` and index number

All the test cases in SWETest are stored inside the test context as interactions in the multivalued child `testCase`. Each of the test cases in the source model maps to a number of test cases in the target model. The target test cases contain interaction fragments that are topologically sorted, i.e., represent linear extensions of the original test case. The name for each target test case is composed of the original name and a running index.

Source element	*Target element*
All `Model:TestSuite:TestContext:Interaction: Lifeline` elements	One `Model:TestSuite:TestContext:Interaction:Lifeline` element
`Lifeline.name`	`Lifeline.name` = 'compositeComponent'
`Lifeline.type` → `TestComponent`	`Lifeline.type` → composite component

Like the test components themselves, their lifelines in the test cases, which can be viewed as their instances, must be joined into one lifeline. The new lifeline gets the name `compositeComponent`. Its type in the target model is a reference that points to the composite component transformed previously.

Source element	*Target element*
`Model:TestSuite:TestContext:Interaction:Message`	`Model:TestSuite:TestContext:Interaction:Message`
`Message.covered` → `Lifeline` of a test component	`Message.covered` → `Lifeline` of composite component
`Message.sourceLifeline` → `Lifeline` of a test component	`Message.sourceLifeline` → `Lifeline` of composite component
`Message.targetLifeline` → `Lifeline` of a test component	`Message.targetLifeline` → `Lifeline` of composite component

Since the messages in the test cases are sent to or from test component lifelines, some references are mapped to references pointing to the new lifeline in the target model. This is a general rule that is valid for both call messages and signal messages whose transformations are defined below.

Source element	*Target element*
`Model:TestSuite:TestContext:Interaction:CallMessage`	`Model:TestSuite:TestContext:Interaction:CallMessage`
`CallMessage.operation` → `Activity` in a test component	`CallMessage.operation` → `Activity` in composite component

Source element	*Target element*
`Model:TestSuite:TestContext:Interaction:SignalMessage`	`Model:TestSuite:TestContext:Interaction:SignalMessage`
`SignalMessage.sourcePort` → `Port` in a test component	`SignalMessage.sourcePort` → `Port` in composite component
`SignalMessage.targetPort` → `Port` in a test component	`SignalMessage.targetPort` → `Port` in composite component

Call messages and signal messages contain references to elements inside test components. The reference to an activity in a call message is transformed into a reference to the corresponding activity in the new composite component. Similarly, the source port and the target port references in a signal message map to ports that are inside the composite component.

3.6 SWETest to TTCN-3 Transformation

SWETest models define test cases that cannot yet be executed. For that reason, an SWETest model must be transformed into executable code of a programming language that is suitable to represent test cases. In this thesis, the testing language TTCN-3[3] was chosen as the target language for the model to text transformation.

Similar to the previous two sections, transformation rules are defined in mapping tables. SWETest elements on the left are opposed to TTCN-3 code samples on the right. The code contains placeholders for data values that are enclosed in angle brackets. The placeholders contain attributes of the source element or a path starting from the source element. They indicate data that is transferred from the model to the code during the transformation. If there is a list of elements involved in a transformation, e.g., a parameter list, then only code for the first element is displayed and the list is indicated by three dots.

SWETest element	*TTCN-3 code*
`Model:TestSuite`	`module <name>{}`

Each test suite in the source model maps to a TTCN-3 module. The name of the test suite becomes the name of the module. A module in TTCN-3 is a container for other elements, e.g., definitions for data types, templates, components, or test cases. Transformation rules for those elements are presented in the following subsections.

Data Definitions

All the data elements in an SWETest model are transformed into suitable definitions in TTCN-3. This includes data elements in the `TestData` container, as well as some elements containing data inside the test context and test cases.

[3]For TTCN-3 code to be executable, it is required that implementations of an execution environment and an SUT adapter are present

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Attribute`	`modulepar{` `<type.name><name> := <defaultValue>;` `...}`

A test context attribute in an SWETest model is visible to all the test cases that are inside the same test context. Such an attribute maps to a module parameter in TTCN-3. The type and the name of the attribute are transferred to the definition of the parameter. If the parameter contains a default value, it is taken to initialize the TTCN-3 parameter.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestData:DataPool:` `Enumeration`	`type enumerated <name>{` `<enumerationLiteral.name>` `(<enumerationLiteral.value>),` `...}`

SWETest enumerations are transformed into corresponding structures in TTCN-3, which are called `enumerated`. The names and the values of the SWETest enumeration literals are used to define fields in the TTCN-3 enumerations. The value in the brackets is only specified, if the optional `value` attribute in the SWETest model is set.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestData:DataPool:` `DataType` with no attributes	`type <redefines> <name>;`

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestData:DataPool:` `DataType` with attributes	`type record <name>{` `<attribute.type> <attribute.name>` `optional,` `...}`

Each data type element in SWETest maps either to a simple `type` definition or to a record. In the first case, the data type does not have any attributes. The value of `redefines` is optional, so it is only used if it is set. The second kind of `DataType` elements, which is transformed to records, contains attributes. Each of the attributes is mapped to a field inside the TTCN-3 record. The keyword `optional` is only set if the SWETest model contains instances of the current data type that do not have a value for the respective attribute or contain a literal null value in that place.

SWETest element	*TTCN-3 code*

`Model:TestSuite:`
`TestData:Signal`
	`type record <name>{`
	`<attribute.type> <attribute.name>`
	`optional,`
	`...}`

Similar to the complex data types above, for each signal, there is a record in the TTCN-3 source code. In this case, the `optional` keyword is produced if there are signal messages in the model that send the signal and do not contain an argument for the attribute or if the attribute has the value literal null.

SWETest element	*TTCN-3 code*

`Model:TestSuite:`
`TestContext:Interaction:`
`SignalMessage`
	`type template <signal.name> <name>{`
	`<signal.attribute.name>:= <argument>,`
	`...}`

The data from a signal message is used to define a TTCN-3 template. The name of the referenced signal is taken as the template type, i.e., it is a record from the transformation rule above. The name of the signal message maps to the name of the template. For each argument that is sent in the message, there is a field in the template that assigns the argument value to an attribute of a signal. If for a certain attribute, there is no corresponding argument in the signal message, or if the argument value is literal null, then the TTCN-3 keyword `omit` is used instead of the argument value.

For reasons of simplification and a better overview, the transformation rules above and those in the subsequent sections show `ValueSpecification` and `DataType` elements as simple text inside TTCN-3 code samples. This is the case for all placeholder paths that end with `value`, `argument`, or `type`. The actual mappings are shown in the two tables below.

SWETest element	*TTCN-3 code*
`...:LiteralString`	`"<value>"`
`...:LiteralInteger`	`<value>`
`...:LiteralBoolean`	`<value>`
`...:LiteralFloat`	`<value>`
`...:LiteralAny`	`?`
`...:LiteralAnyOrNull`	`*`
`...:LiteralNull`	`omit`
`...:EnumerationValue`	`<enumerationLiteral>`

`...:InstanceValue`	`{`
	`<instance.type.attribute.name>:=`
	`<instance.value>,`
	`...}`

Each kind of value specification maps to TTCN-3 code specified by the rules in the table above. A `LiteralString` element is transformed into the value it contains, enclosed in quotation marks. The other elements containing values of simple types just map to their values. The wildcard elements from SWETest become corresponding expressions for wildcards in TTCN-3. Enumeration values contain enumeration literals that are copied to the TTCN-3 output. For each instance value, a block containing initializations of attributes to `value` properties is defined. That value is again one of the value specifications listed in the table above.

SWETest element	*TTCN-3 code*
`Model:TestSuite: TestData:DataPool: DataType` named 'String'	`charstring`
`Model:TestSuite: TestData:DataPool: Enumeration` named 'Verdict'	`verdicttype`
`Model:TestSuite: TestData:DataPool: DataType` named 'Timer'	`timer`
`Model:TestSuite: TestData:DataPool: DataType` named 'Integer'	`integer`
`Model:TestSuite: TestData:DataPool: DataType` named 'Boolean'	`boolean`
`Model:TestSuite: TestData:DataPool: DataType`	`<name>`

Each data type or enumeration in SWETest has a name. Depending on that name, the element is transformed into a string on the right in the above table. The last line of the table states that for each other data type with a different name, the name itself is taken into the TTCN-3 code.

Test Structure

The structure of a test is defined by test components, the SUT, and the communication ports. The mappings involving transformations of those elements are discussed in this subsection.

SWETest element	*TTCN-3 code*

`Model:TestSuite:`
`TestData:PortType`
	`type port <name> message {`
	`inout <signal.name>;`
	`...}`

Each SWETest `PortType` element maps to a message port definition with the same name. For each signal referenced by the port type, there is a record definition in the TTCN-3 port. The keyword `inout` is set by default. However, if the model contains only messages with that signal that are all *received* by a test component, then the keyword `in` is output instead. Similarly, `out` is used if there are only *sent* messages.

SWETest element	*TTCN-3 code*

`Model:TestSuite:`
`TestComponent`
	`type component <name>{`
	`port <port.type.name> <port.name>; ...`
	`timer <timer.name>:= <timer.durationDefault>;`
	`...}`

A test component is transformed into a `type component`. All the ports and timers inside the test component map to port and timer definitions. The duration default value of a timer is optional, i.e., it is only copied to TTCN-3 if it is set in SWETest.

SWETest element	*TTCN-3 code*
–	`type component MainTestComponent {}`

An SWETest model can contain several test components that will be mapped to TTCN-3 components. In TTCN-3, each test case needs a component which it runs on. During the transformation, an empty component with the name `MainTestComponent` is produced, which takes that role for the test cases.

SWETest element	*TTCN-3 code*

`Model:TestSuite:SUT`	`type component SUTInterface {`
	`port <port.type.name> <port.name>;`
	`...}`

The `SUT` element is also transformed into a component. The name of that component is always `SUTInterface`. Each port of the SUT maps to a TTCN-3 port definition inside the component.

General Behavior

SWETest interactions that are not test cases, are used to model behavior supporting the test cases. Those interactions are transformed into altsteps and functions in TTCN-3. The transformation rules defined here are partly reused in the next subsection, where the generation of test cases is described.

SWETest element	*TTCN-3 code*
`Model:TestSuite: TestContext: Interaction` containing only one `alt` combined fragment	`altstep <name> (<parameter.type> <parameter.name>, ...) runs on <lifeline.type.name>{}`

Each interaction that contains only one combined fragment of type `alt` is transformed into an altstep. The name and the possible parameters are taken to specify the name and the parameter definitions for the altstep. The test component whose lifeline is included in the interaction corresponds to the TTCN-3 component that is defined in the `runs on` clause.

SWETest element	*TTCN-3 code*
`Model:TestSuite: TestContext:Interaction: CombinedFragment: InteractionOperand`	`[<condition.expression>] <fragment> {}`

In TTCN-3, an altstep consists of several alt blocks. Hence, each operand in the combined fragment is transformed into an alt block inside the altstep. If the operand contains a condition with a boolean expression, it is transformed into a guard of the alt block. Each fragment inside the operand maps to a TTCN-3 statement inside the alt block. Transformation rules for expressions and fragments are described below.

SWETest element	*TTCN-3 code*
`Model:TestSuite: TestContext: Interaction`	`function <name> (<parameter.type> <parameter.name>, ...) runs on <lifeline.type.name>{}`

Every other interaction of the test context is mapped to a TTCN-3 function definition. The only difference to the transformation above is the keyword `function` and the fact that a function does not have the inner structure of an altstep. All the transformation rules involving the structure of a function are described in the following.

SWETest element	*TTCN-3 code*
`Model:TestSuite: TestContext:Interaction: Lifeline:Default`	`var default <interaction.name>Default := activate (<interaction.name>(<parent.timer>));` `...` `deactivate(<interaction.name>Default);`

If the test component lifeline in the interaction contains a `Default` element, a default activation and deactivation are generated at the beginning and at the end of the function, respectively. The dots here do not have the meaning of repetition, but indicate that there can be other code in that place. For the activation of the default, there is a TTCN-3 `default` variable that is also used to deactivate the same default. If the test component of the lifeline contains a timer, then that timer is placed into the default activation as argument.

Next, the transformation rules for all the possible interaction fragments are described. Fragments correspond to statements that can be contained inside a TTCN-3 function. Those statements define the behavior of the function.

SWETest element	*TTCN-3 code*
`...:Interaction: InteractionUse`	`<interaction.name>(<argument>,...);`

An interaction use defines the call of another interaction. It maps to a call of a TTCN-3 function or altstep. Possible arguments are taken into the argument list.

SWETest element	*TTCN-3 code*
`...:Interaction: SignalMessage` sent by a test component	`<sourcePort.name>.send(<name>);`

Signal messages define the transfer of signals and arguments for the signal attributes between test components and the SUT. Each signal message that is sent from a test component to the SUT is transformed into a `send` statement in TTCN-3. The sending port is the source port of the test component. Since signal messages were also transformed into templates above, the name of the signal message corresponds to the name of a template that is sent to the SUT.

SWETest element	*TTCN-3 code*
`...:Interaction: SignalMessage` sent by the SUT	`<targetPort.name>.receive(<name>);`

The second kind of signal messages are those that are sent out from the SUT and received by a test component. In this case, the message maps to a `receive` statement. Since the behavior in TTCN-3 is defined for test components, the target port is taken as the receiving port in TTCN-3. In SWETest, a target port in a received signal message is optional. If it is not set, then the expression `any port` is output instead.

SWETest element	*TTCN-3 code*
`...:Interaction:` `CallMessage` referencing an activity with a finish action	`stop;`

SWETest element	*TTCN-3 code*
`...:Interaction:` `CallMessage` referencing an activity with a start timer action	`<argument.timer.name>` `.start(<argument.value>);`

SWETest element	*TTCN-3 code*
`...:Interaction:` `CallMessage` referencing an activity with a stop timer action	`<argument.timer.name>.stop;`

SWETest element	*TTCN-3 code*
`...:Interaction:` `CallMessage` referencing an activity with a timeout action	`<argument.timer.name>.timeout;`

The rules above define the transformations for call messages that references an activity to be executed. The result of the transformation is dependent on the contents of the activity. More precisely, the activity can contain one of the four actions specified in the tables above. A finish action defines the stopping of a test component. In TTCN-3, the corresponding statement is `stop`. The other three transformation rules involve timer actions. The call messages are transformed into corresponding TTCN-3 timer statements. In this case, the first argument in SWETest is of type `TimerReference`, whose `timer` property refers to a timer in a test component. The second argument, which is used in the `start` statement, is a `DurationValue`, i.e., it is the duration for the timer in TTCN-3.

SWETest element	*TTCN-3 code*
`...:Interaction:` `CallMessage` with a verdict argument	`setverdict` `(<argument.enumerationLiteral>);`

SWETest element	*TTCN-3 code*
`...:Interaction: CallMessage` with a `NativeCode` argument	`<argument.value>`

In SWETest, there are two kinds of call messages that do not refer to an activity, but have a specific kind of arguments. If there is an argument of type `Verdict`, the call message becomes the setting of a verdict in TTCN-3. The argument is an enumeration value, so its enumeration literal corresponds to the TTCN-3 verdict value. An exception to that occurs, if the value of the literal is `inconclusive`. In that case, it is mapped to `inconc`.

SWETest element	*TTCN-3 code*

`...:Interaction: CombinedFragment` with operator `loop`	`while(<operand.condition.expression>){`
	`}`

SWETest element	*TTCN-3 code*

`...:Interaction: CombinedFragment` with operator `opt`	`if(<operand.condition.expression>){`
	`}`

Combined fragments of types `loop` and `opt` each contain only one operand. They are transformed into the corresponding type of blocks in TTCN-3. The condition of the operand maps to a condition in TTCN-3 syntax. All the other contents of the operand, i.e., interaction fragments that can be again combined fragments, are transformed and placed inside the curly braces of the block.

SWETest element	*TTCN-3 code*

`...:Interaction: CombinedFragment` with operator `alt` guard messages	`alt{`
	`[<operand.condition.expression>]`
	`<operand.fragment>{}`
	`...}`

SWETest element	*TTCN-3 code*

`...:Interaction: CombinedFragment` with operator `alt` and no guard messages	`if(<operand.condition.expression>){ } else if`
	`(<operand.condition.expression>){`
	`}...`
	`else {`
	`}`

An `alt` fragment can map either to an `alt` or to an `if` statement. A *guard message* can be either an incoming signal message or a call message that triggers a timeout. It must be the first fragment of an interaction operand. In that case, this message is treated like an additional guard and the enclosing combined fragment is transformed into an `alt` statement with the first fragment as the additional guard of each `alt` block. If no such guard message is present, then the combined fragment is mapped to an `if-else` construct.

SWETest element	*TTCN-3 code*
`...:Greater`	`<leftOperand> > <rightOperand>`
`...:Less`	`<leftOperand> < <rightOperand>`
`...:Equals`	`<leftOperand> = <rightOperand>`
`...:GreaterOrEquals`	`<leftOperand> >= <rightOperand>`
`...:LessOrEquals`	`<leftOperand> <= <rightOperand>`
`...:NotEquals`	`<leftOperand> != <rightOperand>`
`...:Not`	`not(<operand>)`
`...:And`	`(<leftOperand>) and (<rightOperand>)`
`...:Or`	`<leftOperand> or <rightOperand>`
`...:Else`	`else`

The table above defines transformation rules for all the possible boolean expressions and for the `else` statement. Each SWETest expression corresponds to a TTCN-3 boolean guard containing operands and the proper operator. Those transformations are applied to the placeholders ending with `expression` in other rules.

Test Cases

All the interactions of the test context that are inside the `testCase` property are transformed to TTCN-3 test cases and supporting functions. A test case in TTCN-3 is used to initialize and start the behavior of test components. The behavior of each component is always defined in an extra function.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Interaction` as `testCase`	`function <name>_<lifeline.name>` `(<parameter.type> <parameter.name>, ...) runs` `on <lifeline.type.name>{}`

Each SWETest test case maps to several TTCN-3 functions. Each function contains the behavior of one of the test components involved in the test case. The instance of that test component is represented by a lifeline, whose name is used to compose different names for the functions. Also, the type of the lifeline, i.e., the test component itself, maps to the TTCN-3 component definition in the `runs on` clause.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Interaction` as `testCase`	`testcase <name>` `(<parameter.type> <parameter.name>, ...) runs on MainTestComponent system SUTInterface {}`

The test cases are also mapped to TTCN-3 test case definitions, which run on the predefined main test component. The `system` clause always defines the predefined interface to the SUT. Test cases do not contain any other behavior than the statements described in the following three transformation rules.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Interaction:` `Lifeline` of a test component	`var <type.name> <name>:= <type.name>.create;`

For each test component lifeline in the interaction, there is a variable definition with the assignment of a component instance to it. This means that concrete components named like the lifelines are created. The `TestComponent` element that is referenced in the lifeline supplies the name for the variable type.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Interaction:` `SignalMessage`	`map(<sourceLifeline.name>: <sourcePort.name>, system:<targetPort.name>);`

In TTCN-3, ports of the test components must be mapped to ports of the SUT in order to send messages between those ports. For this, signal messages from the SWETest model are selected that contain different port pairs and are transformed into `map` statements. In the table above, the case for a signal message is shown, where it is sent from a test component to the SUT. In the other direction, source and target will be swapped.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Interaction:` `Lifeline` of a test component	`<name>.start(` `<parent.name>_<name>` `(<parent.parameter.name>, ...)` `)`

The last possible statement in a generated test case is the starting of a component behavior. The previously created component variable is used to start the previously created component function. If there are any parameters in the parent interaction of the lifeline, then they are passed to the function.

Test Control

Both in SWETest and in TTCN-3, there is a control part that is used to execute the test cases. In SWETest, it consists of test case calls whose verdicts can be stored, logged, and evaluated for simple conditional branching. Transformation rules for mapping a test control part of an SWETest model to TTCN-3 source code constructs are defined in the following.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Activity` as `testControl`	`control {}`

The activity in the test context that is inside the property `testControl` is transformed into a `control` part of the TTCN-3 module. The contents of the activity map to TTCN-3 structures inside the `control` part. Transformation rules for those structures are described next.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Activity:` `ObjectNode`	`var <type> <name>;`

For each object node with different names in the test control activity, there is a variable definition in TTCN-3. The type and the name of the object node are are used as the name and the type for the variable.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Activity:` `CallBehaviorAction`	`<outgoing.target.name>:=` `execute(<interaction.name>);`

Call behavior actions map to TTCN-3 `execute` statements that start the test case whose name is the same as the referenced interaction in SWETest. In SWETest, an object node might follow the call behavior action, which means that the verdict of the called test case will be stored in it. In this case, the `execute` statement is assigned to a variable that has been declared in the above transformation.

SWETest element	*TTCN-3 code*
`Model:TestSuite:` `TestContext:Activity:` `LogAction`	`log("<value>");`

An SWETest log action corresponds to a `log` statement in TTCN-3. The value stored in the log action becomes the string value to be logged in TTCN-3.

SWETest element	*TTCN-3 code*

`Model:TestSuite:`
`TestContext:Activity:`
`DecisionNode`
	`if (<outgoing.condition.expression>){`
	`}`
	`else if`
	`(<outgoing.condition.expression>){`
	`}...`

Decision nodes define simple branching execution that depends on conditions in outgoing edges. Decision nodes are transformed into `if` statements that can be followed by `else if` blocks. The conditions in outgoing edges map to TTCN-3 conditional statements.

4 Implementation

The concepts that are described in the previous chapter are implemented with the help of several tools, which are based on the Eclipse platform [5]. The primary tool used is openArchitectureWare (oAW) [12]. It executes the model to model and model to text transformations. SWEUTP models (which are the input to oAW in the first transformation) are created with the UML tool MagicDraw [9]. The models are stored as XMI files in the EMF UML2 format, which can be processed by oAW. This is done with the help of the UML2 plug-in that is part of the Model Development Tools (MDT) of Eclipse [4]. Constraint checking and transformations are coordinated in oAW workflows that contain workflow components for each performed action. Definitions for model to model transformations are specified in Xtend. More complex behavior like expression parsing and topological sorting is implemented in Java. The grammar and the parser for boolean expressions were generated with ANTLR [1]. The SWETest metamodel, which was created with EMF Ecore editors, exists as an XMI file and as generated Java classes. The additional Java format is necessary for the algorithms that are implemented in Java. The model to text transformation that generates TTCN-3 code is implemented as a number of templates in Xpand. The structure of this chapter is as follows:

- In Section 4.1, it is described how the OCL constraints are used to validate an SWEUTP model.
- The two implementations of model to model transformations are described in Section 4.2 and Section 4.3.
- The generation of TTCN-3 code is explained in Section 4.4

4.1 Constraining SWEUTP

There is a number of exemplary OCL constraints that are used to validate an SWEUTP model, i.e., to check if the model is suitable for the transformation into an SWETest model. The constraints are stored in a file which is structured as shown in the following listing. For each constraint, there is a comment representing a short description, followed by the constraint expression.

```
-- Signal messages must have names
Message.allInstances()->select(messageSort=MessageSort::asynchSignal)
   ->forAll(name<>'')
```

This file is read by a workflow component that parses it and extracts single constraints and the descriptions belonging to each constraint. Each constraint is then evaluated on the model, which is accessible from the workflow component. The Eclipse OCL implementation [4] is used for the evaluation. The next listing illustrates Java code that evaluates one constraint.

```
 1 IOCLFactory<Object> oclFactory = new UMLOCLFactory(context);
 2 ModelingLevel modelingLevel = ModelingLevel.M2;
 3 OCL<?, Object, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?> ocl;
 4 ocl = oclFactory.createOCL(modelingLevel);
 5 OCLHelper<Object, ?, ?, ?> helper = ocl.createOCLHelper();
 6 try {
 7   modelingLevel.setContext(helper, context, oclFactory);
 8 
 9   switch (modelingLevel) {
10   case M2:
11     OCLExpression<Object> parsed = helper
12         .createQuery(expression);
13     Object results = ocl.evaluate(context, parsed);
14     if (results instanceof Boolean) {
15       boolean bool = (Boolean) results;
16       if (bool == false)
17         issues.addError("OCL Check failed: " + currentDescription);
18     }
19     break;
20   case M1:
21     // not implemented
22     break;
23   }
24 
25 } catch (ParserException e) {
26   issues.addError(e.getMessage());
27 }
```

The `context` object contains the UML model that will be validated. First, a factory is initialized with that object (line 1). The enumeration object named `modelingLevel` is set to `M2`, as all the OCL constraints operate on OMG's M2-level, i.e., on metamodel elements (line 2). An `OCL` object is built using the first two objects and a corresponding helper is created that will parse the string containing the OCL expression (lines 3 – 5). The method `setContext` in line 7 associates the given context to the helper.

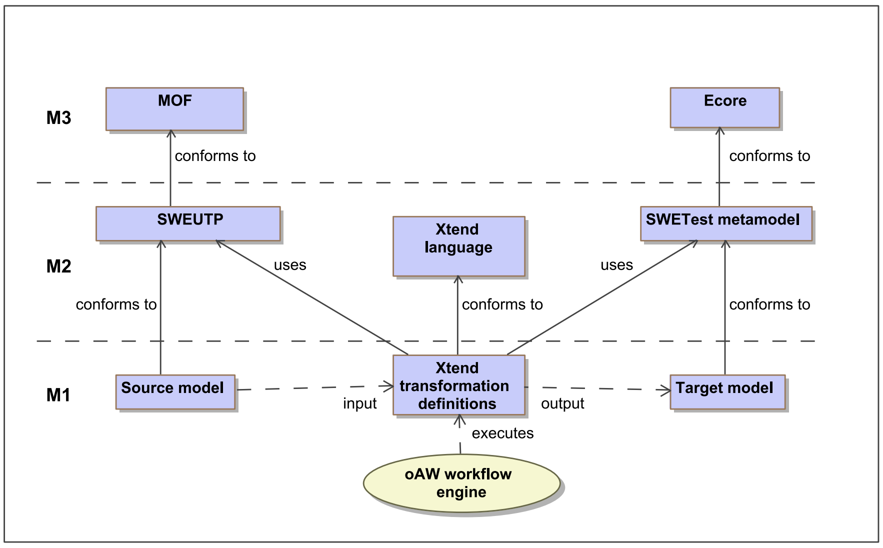


Figure 4.1: Model to model transformation with openArchitectureWare

Since the modeling level is always M2, the first `case` block is executed. The variable `expression` is a string containing the OCL constraint. In line 11, the helper parses it and produces an `OCLExpression` object. Now, the `OCL` object can evaluate the parsed expression on the UML model contained in `context` (line 13). If the evaluation returns `false`, an error is generated containing the description of the constraint. The error message will be printed in the console.

```
<component
    class="workflow.components.oclvalidator.UMLOCLValidatorComponent">
  <oclConstraintsFile value="${oclfile}"/>
  <modelSlot value="umlmodel"/>
</component>
```

The workflow component that evaluates the OCL constraints is started in a workflow as shown in the listing above. The file referenced in the `oclfile` property is the file that contains the constraints. The slot named `umlmodel` holds the whole model that is supposed to be validated.

4.2 SWEUTP to SWETest Transformation

Model to model transformations are defined in Xtend. In Figure 4.1, an overview of the transformation from SWEUTP to SWETest is shown. An SWEUTP source model serves as input to Xtend definitions that transform it into an SWETest target model using both metamodels. The definitions are executed by the workflow engine of oAW. In this section, first some examples of the implemented transformation definitions are described. After that, the process of invoking the definitions from a workflow is explained.

Transformation definitions in Xtend are called *extensions*, because they can be used not only to transform models, but also to extend a present model without creating a new one. A special form of extensions, `create` extensions, are used here to instantiate new objects of SWETest metamodel elements.

```
create Model transform( uml::Model uml_model ):
  testSuite.addAll(
    uml_model.allOwnedElements().typeSelect(uml::Class).
    select(el|el.getAppliedStereotypes().exists(s|s.name ==
    "TestContext")).createTestSuite() );
```

In the listing above, a new SWETest `Model` object is created and the extension named `transform` is executed. The object is available in the extension through the reference `this`, which is omitted here. The information contained in the only argument of the extension, which contains the SWEUTP source model, can be accessed inside the extension. The multivalued containment reference `testSuite` of the `Model` object is filled with elements that are returned by the statement inside the brackets. This statement analyzes the SWEUTP model for certain elements. It selects only classes that have the stereotype `TestContext` and executes the extension `createTestSuite` for each of them. The whole statement produces a list of `TestSuite` objects that are added to `testSuite`. In the following listing, the creation of a test suite is defined.

```
create TestSuite createTestSuite(uml::Class uml_testContext):
  setName( uml_testContext.package.name )->
  setTestData(uml_testContext.package.
    createTestData(uml_testContext))->
  sut.addAll( uml_testContext.getAllAttributes().
    select(a|a.getAppliedStereotypes().exists(s|s.name == "SUT")).
    createSUT(this))->
  testComponent.addAll( uml_testContext.part.type.
    typeSelect(uml::Class).select(a|a.getAppliedStereotypes().
    exists(s|s.name == "TestComponent")).createTestComponent(this) )->
  arbiter.addAll( uml_testContext.getAllAttributes().type.
    typeSelect(uml::Class).select(a|a.getAppliedStereotypes().
    exists(s|s.name == "Arbiter")).createArbiter(this) )->
  arbiter.addAll( uml_testContext.ownedAttribute.
    select(attr|attr.type.name == "Arbiter1"
    || attr.type.name == "Arbiter").createArbiter(this) )->
  setTestContext(uml_testContext.createTestContext(this) );
```

To set all the properties of a test suite, the data in the SWEUTP test context is used, which is a UML class. The name and the test data are taken from the package that the class is in (lines 2 – 4). The SUT is transformed on lines 5 through 7. All the attributes of the test context that contain the stereotype `SUT` are selected and passed to the extension `createSUT` (which is not shown here). Similarly, test components are created by selecting the types of the parts in the test context that are stereotyped as `TestComponent`. For each of them, the extension `createTestComponent` is called. Since there are two kinds of arbiter types possible in SWEUTP – a user-defined class or the predefined interface – there are two corresponding statements that create SWETest arbiters. MagicDraw exports the predefined interface as `Arbiter1`, therefore that name is also checked in the second statement (lines 14 – 16). Finally, the SWEUTP test context is passed to an extension to create an SWETest test context.

```
1  create Interaction createInteraction(uml::Interaction uml_interaction,
2  TestSuite swe_suite):
3    uml_interaction.specification != null?
4      setName(uml_interaction.specification.name)
5      :setName(uml_interaction.name)->
6    lifeline.addAll(uml_interaction.lifeline.
7      createLifeline(swe_suite))->
8    fragment.addAll(uml_interaction.fragment.
9      select(f|f.metaType.name == "uml::MessageOccurrenceSpecification"
10     && ((uml::MessageOccurrenceSpecification)f).message.
11     receiveEvent == f || f.metaType.name == "uml::CombinedFragment"
12     || f.metaType.name == "uml::InteractionUse").
13     createInteractionFragment(this, swe_suite))->
14   fragment.addAfter(this)->
15   uml_interaction.specification != null?
16     parameter.addAll(uml_interaction.specification.
17       ownedParameter.createParameter(swe_suite.testData.dataPool))
18   : parameter.addAll(uml_interaction.ownedParameter.
19       createParameter(swe_suite.testData.dataPool));
```

Test cases are represented as interactions inside the test context. The extension above creates an SWETest interaction using a UML interaction. The second argument, `swe_suite`, is part of the target SWETest model and is used in called extensions to manipulate or reference already existing model elements. In lines 3 – 5, the name of the newly created `Interaction` object is set. The name of the referencing operation is preferred. All the lifelines are just passed through to another extension (lines 6 and 7). To create SWETest interaction fragments (lines 8 – 13), only certain UML fragments are selected. The first part of the boolean expression that involves `MessageOccurrenceSpecification` selects only those specifications that occur on receiving lifelines. Those specifications will be transformed into call messages or signal messages. Additionally, all the combined fragments and interaction uses are selected. There are several extensions named `createInteractionFragment`, which are called dependent on the type of the first parameter, i.e., the fragment.

In line 14, an extension is called for each SWETest fragment. The extension finds out which other fragment is executed directly before the argument fragment. The result is a partial order defined on all the fragments of an interaction. This information will be used during the next model to model transformation in the next section to compute topological sorts. The conditional block at the end of the extension transforms either operation parameters or interaction parameters into SWETest parameters.

There are three workflow components that are responsible for transforming an SWEUTP model into an SWETest model: one that reads the source file, another that performs the transformation, and a third one that writes the resulting model to a target file. The configurations for those components are discussed in the following.

```
<component class="oaw.emf.XmiReader">
  <modelFile value="${inputuml}"/>
  <outputSlot value="umlmodel"/>
</component>
```

The workflow component with the type `XmiReader` reads a file in the EMF UML2 format. The model contained in the file is put into the slot `umlmodel`. After that, the model is available to other workflow components.

```
<component class="oaw.xtend.XtendComponent" skipOnErrors="true">
  <metaModel id="mm" class="oaw.type.emf.EmfMetaModel">
    <metaModelPackage value="swetest_mm.Swetest_mmPackage"/>
  </metaModel>
  <metaModel class="oaw.type.emf.EmfMetaModel"
    metaModelPackage="org.eclipse.emf.ecore.EcorePackage"/>
  <metaModel class="oaw.uml2.UML2MetaModel"/>
  <invoke value="${xtendfile}::transform(umlmodel)"/>
  <outputSlot value="swemodel"/>
</component>
```

After loading the necessary metamodels, the component above executes the `transform` extension, which is the top level extension in the file containing the transformation definitions. The SWEUTP model that is inside the slot `umlmodel` is passed as an argument to the extension. The returned SWETest model is stored in the output slot `swemodel`.

```
<component class="org.eclipse.mwe.emf.Writer">
  <useSingleGlobalResourceSet value="true"/>
  <modelSlot value="swemodel"/>
  <uri value="${outputxmi}"/>
</component>
```

Finally, the SWETest model is written to a file by the `Writer` component. The slot of the model is the same as the output slot in the previous listing. The output path for the file is given here by the `outputxmi` property.

4.3 SWETest to SWETest Transformation

This kind of transformation is an *in-place* transformation, because there is only one model that is modified directly. The intention of this transformation is to generate test suites that contain test cases with deterministic behaviors. Therefore, the resulting model contains only one test component that communicates with the SUT. The behavior fragments of each test case that involves more than one component in the original model, are sorted topologically. There is a new test case for each linear extension of the partial order in the original test case.

```
performTopologicalSorting( Model this ):
  testSuite.addNewElements()->
  testSuite.testContext.testCase.addTestCase()->
  testSuite.removeOldElements();
```

The extension called `performTopologicalSorting` starts the transformation process. At the beginning, the `Model` argument contains the original model. Some new elements and test cases are added to it and elements that are not needed anymore are deleted from it.

```
create TestComponent createTestComponent(
List[TestComponent] old_components ):
  setName("CompositeComponent")->
  timer.addAll(old_components.timer.createTimer())->
  port.addAll(old_components.port.createPort())->
  activity.addAll(old_components.activity.createActivity());
```

One of the elements that are added to the model is a test component named `Composite-Component`, which replaces all the other test components. Timers, port, and activities of the original test components are copied into the new component. All the references in the model referring to the original elements are changed to point to the new elements.

```
addTestCase( Interaction old_testCase ):
  ((TestContext)old_testCase.eContainer).testCase.addAll(
    getTopologicalSorts(old_testCase).integerList.
    createTestCase(old_testCase) );
```

The extension above is called once for each test case in the original model. Each time, a number of new test cases is added to the test context. The extension `getTopologicalSorts` is mapped to a Java method that implements the extended version of Kahn's algorithm, which is described in the "Foundations" chapter. The Java method returns several linear extensions represented as integer lists. Each list is ordered corresponding to the respective topological sort. For each of those lists, an interaction is created by the extension `createTestCase`, which is described below.

```
create Interaction createTestCase( IntegerList numbers,
Interaction old_testCase ):
  let testCaseNumber = ((ListOfLists)numbers.eContainer).integerList.
    indexOf(numbers): null->
  setName(old_testCase.name + testCaseNumber)->
  lifeline.addAll(old_testCase.lifeline.select(ll|ll.type.metaType.
    name != "swetest_mm::testElements::TestComponent").
    createNonComponentLL(testCaseNumber))->
  lifeline.add(old_testCase.lifeline.select(ll|ll.type.metaType.
    name == "swetest_mm::testElements::TestComponent").
    createComponentLL(testCaseNumber))->
  fragment.addAll(numbers.item.handleFragment(old_testCase,
    testCaseNumber));
```

The name of each new test case id constructed from the old name and the index value of the position of the integer list inside its parent, i.e., the sequence number of the linear extension. New lifelines are created using the data of the old ones. The extension `handleFragments`, which is called for each number in the linear extension, returns the fragment that is in that position. Therefore, the new `fragment` list contains fragments ordered as specified in the `numbers` list.

```
    <component class="org.openarchitectureware.emf.XmiReader">
      <modelFile value="${inputfile}"/>
      <outputSlot value="swemodel"/>
      <firstElementOnly value="true"/>
    </component>
```

```
  <component class="oaw.xtend.XtendComponent" skipOnErrors="true">
    <metaModel id="mm" class="oaw.type.emf.EmfMetaModel">
      <metaModelPackage value="swetest_mm.Swetest_mmPackage"/>
    </metaModel>
    <metaModel class="oaw.type.emf.EmfMetaModel"
      metaModelPackage="org.eclipse.emf.ecore.EcorePackage"/>
    <invoke
      value="${xtendfile}::performTopologicalSorting(swemodel)"/>
  </component>
```

The two listings above show the configuration of the workflow components that are needed to perform the transformation. The first component reads the model from a file and puts it into the slot `swemodel`. The second component invokes the start extension in the Xtend file and passes the model as argument. The writing of the result model to a file is the same as in the previous section.

4.4 SWETest to TTCN-3 Transformation

During the model to text transformation, TTCN-3 code is generated. The transformation is implemented as Xpand templates. Some example templates and the integration of the transformation into a workflow are described in this section.

```
1  <<DEFINE ttcn3Module FOR TestSuite>>
2  <<FILE name+"_gen.ttcn3"->>
3  module <<name>> {
4  <<EXPAND moduleparams FOR testContext->>
5  <<EXPAND dataTypes(this) FOREACH testData.dataPool.dataType->>
6  ...
7  <<FOREACH testContext.testCase AS tc->>
8    <<EXPAND componentBehaviour(tc) FOREACH tc.lifeline.
9      select(l|l.type.metaType.name ==
10     "swetest_mm::testElements::TestComponent")>>
11 <<ENDFOREACH>>
12 <<EXPAND testCases(this) FOREACH testContext.testCase->>
13 <<IF testContext.testControl != null->>
14 <<EXPAND control FOR testContext.testControl->>
15 <<ENDIF->>
16 }
17 <<ENDFILE>>
18 <<ENDDEFINE>>
```

The main template shown in the listing above generates a TTCN-3 file for each test suite in an SWETest model. It produces a module with the name of the test suite and calls directly or indirectly all the other templates. Each of the called templates generates a certain part of the TTCN-3 module using data of some elements from the SWETest model. In lines 7 through 11, the template called `coomponentBehavior` is called for each component lifeline in each test case. Each call generates a TTCN-3 function that encapsulates the behavior of one component. The template is described below.

```
<<DEFINE componentBehaviour(Interaction inter) FOR Lifeline->>
 function <<inter.name>>_<<name>>
   (<<EXPAND param FOREACH inter.parameter>>)
   runs on <<type.name>> {
   <<FOREACH this.defaults AS def->>
   var default <<def.interaction.name + "Default">> :=
     activate(<<def.interaction.name>>(<<IF def.interaction.
     parameter.size > 0>><<((TestComponent)this.type).timer.
     first().name>><<ENDIF>>));<<"\n"->>
   <<ENDFOREACH->>
   <<EXPAND frag(this, "\t") FOREACH inter.fragment->>

13     <<FOREACH this.defaults AS def->>
14     deactivate(<<def.interaction.name + "Default">>);<<"\n"->>
15     <<ENDFOREACH->>
16   }
17 <<ENDDEFINE>>
```

Since it is possible that a test component participates in different test cases, the generated name of the function is prefixed with the interaction name. If the lifeline contains defaults, then default activations are generated at the beginning and default deactivations at the end of the function body. If the called default interaction, which in TTCN-3 is an altstep, has parameters, then the call of the altstep gets a timer argument. In line 11, all the interaction fragments, i.e., signal messages, call messages, combined fragments, and interaction uses, are transformed into corresponding TTCN-3 structures.

```
1    <component class="org.openarchitectureware.xpand2.Generator">
2      <metaModel
3        class="org.eclipse.m2t.type.emf.EmfRegistryMetaModel"/>
4      <expand
5       value="${templatefile}::ttcn3Module FOREACH swemodel.testSuite"/>
6      <outlet path="${outputttcn}">
7       <postprocessor
8       class="workflow.postprocessors.ttcn3beautifier.TTCN3Beautifier"/>
9      </outlet>
10   </component>
```

The Xpand template definitions are executed by a workflow component. In lines 4 and 5 in the above listing, the root template `ttcn3Module` that is contained in a file is called for each test suite of the SWETest model that is inside the slot `swemodel`. The path for the generated TTCN-3 file is given by the property `outputttcn`. Additionally, a TTCN-3 beautifier is referenced, which formats the generated code.

5 Case Study

In this chapter, an example for the modeling of test cases, transformation to an intermediate model, as well as the code generation for a test language is described. First, a simple communication protocol called Inres is presented. The Inres protocol allows for one entity to connect to a second entity and send data packets to it through a medium. The second entity can answer with receive acknowledgments and then close the connection.

The behavior of the Inres protocol is tested by test cases, which are modeled in SWEUTP and presented in the subsequent section. Two test components replace two entities of the Inres protocol and exchange data with the SUT, which represents the other Inres entities. The structure of the test suite as well as the data that is exchanged are shown and explained in detail.

Transforming the SWEUTP model with the tools implemented in this thesis produces an SWETest model and in the second step a TTCN-3 code file. In the last section of this chapter, the SWETest model and the generated code are presented.

5.1 Inres Protocol

The Inres protocol [25] is a simple communication protocol which can be used for black-box testing of distributed systems. It is not a real-life protocol, but was rather specified to be used in education and research. The protocol transfers data from one client entity to another.

In Figure 5.1, the structure of Inres is shown. The two entities Initiator user and Responder user are the clients that use the Inres service. They can exchange data by accessing the interface of the Inres service called ISAP (Inres Service Access Point). The clients can only see this interface and it is not important to them how the service is implemented. The two Inres service entities Initiator and Responder send data units to each other using the MSAP (Medium Service Access Point) interface of the underlying Medium service. No further structure of the Medium is specified. It is only known, that messages can get lost, but not corrupted or switched. Whenever a data unit comes in at one MSAP, the Medium service transmits it to the opposite MSAP.

MSAP and ISAP offer service primitives which are used by the Inres and client entities to communicate with each other. For example, the Initiator user can use the service primitive

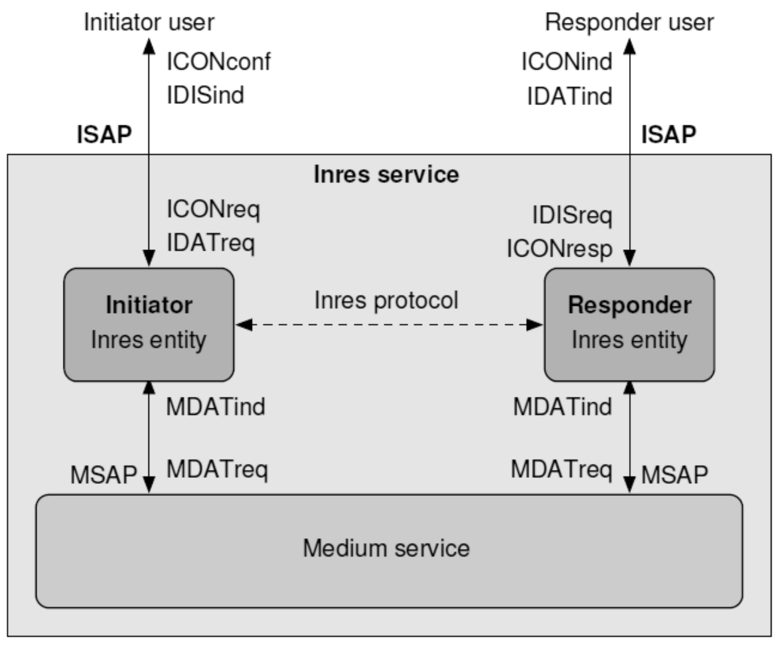


Figure 5.1: The structure of the Inres protocol [31]

ICONreq (stands for ISAP Connection Request)[1] from the Initiator to request a connection. The Inres service can be used by clients for the following actions:

- The Initiator user can establish a connection to the Responder user;
- The Initiator user can send data to the Responder user;
- The Responder user can release the current connection.

The ISAP offers a different service primitive for every possible action that can be executed. In contrast to that, the MSAP interfaces have only two primitives, which are used universally for all kinds of actions: MDATreq to request a data transfer on the medium and MDATind to indicate incoming data to an entity of the higher layer. The sort of the message contents is distinguished by the first argument given to the primitive.

- If a connection is requested, 'CR' is sent through the medium.
- A connection confirmation is indicated by 'CC'.
- 'DT' is used for data transfers.
- Whenever an acknowledgment for a received data unit is sent, an 'AK' attribute is added.
- A 'DR' is sent to indicate a disconnection request.

The establishment of a connection is performed as indicated in Figure 5.2. First, the information that the Initiator user wishes to make a connection is passed through all the Inres service entities to the Responder user. Along the way, a service primitive of every service entity is called, whereupon the entity reacts by calling the right primitive of another entity, thus propagating the information. When the Responder user gets the connection request, it sends out an acknowledgment, which is forwarded back to the Initiator user.

While connected, the Initiator user can send data units to the Responder user. The Initiator user passes the data to the Initiator, which calls the MSAP primitive MDATreq. As described above, the primitive is called with the argument 'DT'. In addition to this, a sequence number is added, which can be used to detect switched messages. Finally, the delivered data unit is appended and the message is sent to the other end of the medium. The Responder receives all the data through the service primitive MDATind. The data unit from the Initiator user is sent to the Responder user. The sequence number, in contrast, is

[1]The names of the service primitives are acronyms for the according actions which they represent. The parts of the words have the meanings: I = ISAP, M = MSAP, CON = connection, DIS = disconnection, DAT = data, conf = confirmation, ind = indication, req = request, resp = response

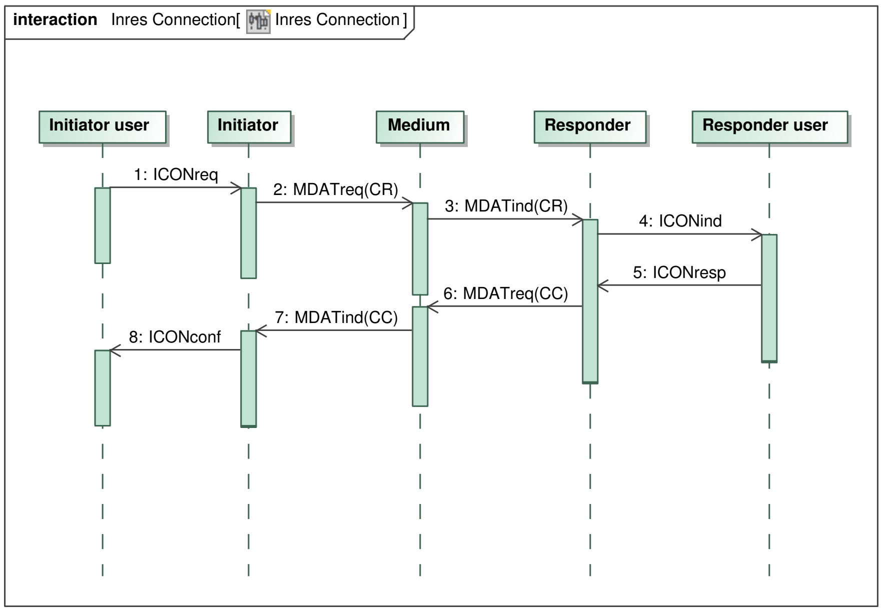


Figure 5.2: Inres connection establishment

extracted and together with an 'AK' attribute sent back to the Initiator. The acknowledgment is used internally by the Inres protocol to indicate message arrivals.

If the Responder user wishes to close the connection, it uses the ISAP service primitive IDISreq. The Initiator user gets this information through the primitive IDISind. The Medium transfers the disconnection request as a 'DR' argument.

5.2 Modeling Test Cases for Inres

The test cases are based on the InresDistributed example[2] [31]. There are two test components that replace the Initiator user and the Responder. The Initiator and the Medium are the services that are tested and hence they are the SUT. The Responder user is not part of the test system. By stimulating the SUT, i.e., by calling the right primitives on the right

[2]In the InresDistributed example only the 'data transfer' test case is implemented. A preamble function is used to establish a connection and a postamble function closes the connection. Here, the connection is established and closed by own test cases. Moreover, the 'data transfer' test case is here simplified, since not all the features of TTCN-3 used in InresDistributed can be modeled and transformed in the implementation of this thesis

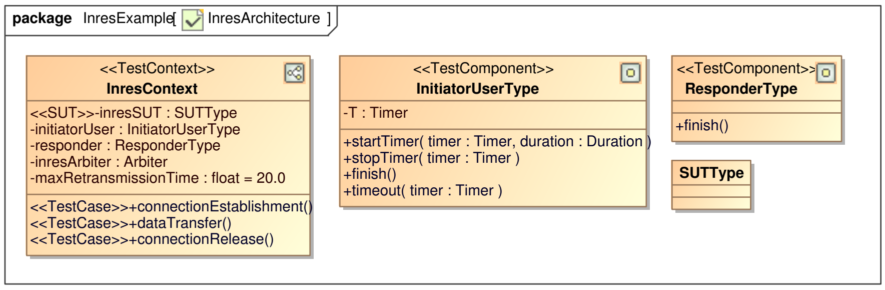

package InresExample[InresArchitecture]
<<TestContext>>
InresContext
<<SUT>>-inresSUT : SUTType
-initiatorUser : InitiatorUserType
-responder : ResponderType
-inresArbiter : Arbiter
-maxRetransmissionTime : float = 20.0
<<TestCase>>+connectionEstablishment()
<<TestCase>>+dataTransfer()
<<TestCase>>+connectionRelease()
<<TestComponent>>
InitiatorUserType
-T : Timer
+startTimer(timer : Timer, duration : Duration)
+stopTimer(timer : Timer)
+finish()
+timeout(timer : Timer)
<<TestComponent>>
ResponderType
+finish()
SUTType

Figure 5.3: Inres test architecture

access points, the test components establish a connection, exchange some data, and then close the connection. If the SUT reacts as expected, then the verdict is set to `pass`, otherwise the verdict can be `fail` or `inconclusive`.

Test Architecture

The Inres test architecture (Figure 5.3) contains a test context, two types for test components, and one type for the SUT. The test context holds one attribute for each of the types. These attributes are parts in the test configuration and are represented by lifelines in the test cases, described below. The attribute `inresArbiter` is used in the test cases to set the verdicts of the test components. The float attribute `maxRetransmissionTime` stores a simple value. The name of this attribute is used by the test components to specify the waiting time for a message. The three test case operations hold references to interactions with the same names, which implement the actual test cases.

The test component type `InitiatorUserType` contains a timer `T` and operations to handle it. The operations point to activities containing the respective UTP actions. The operation `finish()` refers to an activity with the `FinishAction` inside it.

Test Configuration

The test configuration for the modeled test cases is displayed in Figure 5.4. The ISAP and the MSAP interfaces are modeled as pairs of ports which are connected by connectors. The Inres service primitives are modeled by signals, that are sent in messages between the ports (cf. Figure 5.5). The initiator user can communicate with the SUT through its ISAP port. Outgoing messages are immediately transported to the connected ISAP port of the SUT. In reverse, messages sent by the SUT to its ISAP port are forwarded to the initiator user's port as incoming messages. Similarly, the messages are transferred between the two MSAP ports of the SUT and the responder.

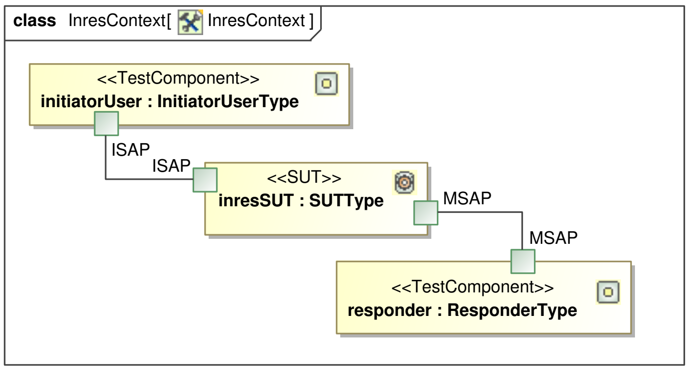

class InresContext[InresContext]
<<TestComponent>>
initiatorUser : InitiatorUserType
ISAP
ISAP
<<SUT>>
inresSUT : SUTType
MSAP
MSAP
<<TestComponent>>
responder : ResponderType

Figure 5.4: Inres test configuration

Test Data

All the required Inres service primitives and the data units passed through them are modeled as classifiers and instances (Figure 5.5). For each primitive there is a corresponding signal which can be sent in a signal message from a port to another port. `InresSAP` and `MediumSAP` are types for the actual ports ISAP and MSAP. The port types define with associations, which signals can be sent to or received from the ports of the respective type.

Some signals contain attributes of user-defined types which correspond to the parameters that are contained in the according primitives. The IDATreq primitive in Inres always contains a data unit to be sent to the responder user. The data type of this unit is not specified in Inres. In the test case modeled here, the unit is represented by the attribute `iData` of the signal IDATreq. Its type is the user defined type `UserPDU`, which is derived from `float`. So, floating point data units can be sent out to the SUT. The other signal attribute is `mData` in the signals MDATreq and MDATind. In Inres, it is possible to pass up to three arguments to the corresponding primitives: The type of the data (e.g., 'DT' for data transfer), a data unit, and a sequence number. Here, those three arguments are modeled as one user defined data type `InresPDU` which is a structured data type and contains three attributes corresponding to the Inres arguments. The data unit field `iData` is again of the `UserPDU` type since it only forwards the data units from the initiator user. For the other two fields, there are two enumerations: `InresPDUType` and `SequenceNumber`. `InresPDUType` contains all the possible values that also can be used in Inres. Since only two data units will be sent in the test case, the possible values for a sequence number are `zero` and `one`.

The actual data that will be sent in the signals MDATreq and MDATind is modeled as UML instances (Figure 5.5). Those will be used for communication between the SUT and the responder test component.

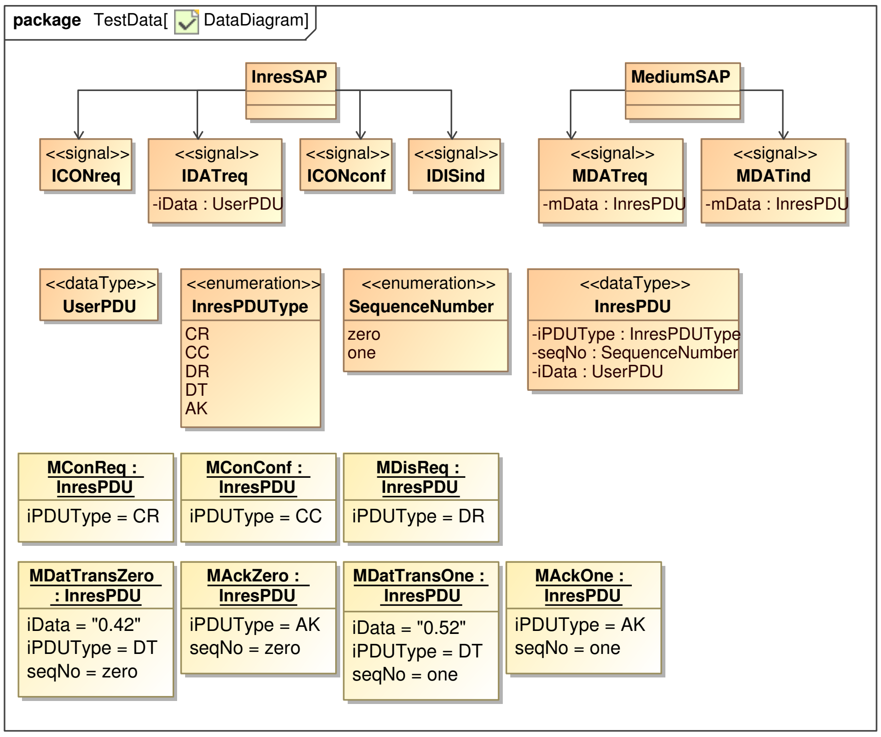


Figure 5.5: Inres test data

- MConReq, MConConf, and MDisReq are used for connection request, connection confirmation, and disconnection request, respectively. The iPDUType fields are set to the respective enumeration values which correspond to the Inres arguments.
- MDatTransZero and MDatTransOne represent the transmission of two data units. The first one has the sequence number zero and transmits the data '0.42'. '0.52' is sent by the second one, which contains the sequence number one.
- Finally, MAckZero and MAckOne are used to send acknowledgements for the two received data units. In addition to the argument 'AK', those contain the respective sequence numbers, which are the same as in the incoming signals with the data units.

Test Control

An interaction overview diagram gives an overview of the test case execution (Figure 5.6). The start and the finish of the execution are logged. A test case called connectionEstablishment is called first. The verdict of the test case is stored in the object node myVerdict. The following decision node has two outgoing edges: the one with the guard is taken, if the value in the previous object node is pass, otherwise the other one. So, if the first test case returns pass, the second test case named dataTransfer is executed. Its verdict is again put in an object node and forwarded to the next decision node (that is also a merge node at the same time). Again, if the verdict is pass, the last test case, connectionRelease, is called. The third test case is only executed, if the other two both returned pass. This dependency between the test cases is modeled, because for the second and the third test case certain states must be present. The data can only be transferred after the connection is established. The connection can only be closed if the first two test cases execute with no errors.

Test Cases

The test cases are modeled as interactions. The lifelines represent the SUT, the test components, and the arbiter. In the following, only those elements are addressed, so 'the SUT' means here 'the lifeline representing the SUT'.

In the test case connectionEstablishment, the opening of a connection is simulated (Figure 5.7). The initiator user starts its timer T via a call message to self with the duration stored in the attribute maxRetransmissionTime. It then sends a signal message containing the ICONreq signal to the port ISAP of the SUT. If after that a message with the ICONconf signal comes in on its own ISAP port, then the initiator user considers the test case as passed. It stops the running timer and calls the operation setVerdict in the arbiter with the argument pass.

When the SUT receives the connection request message from the initiator user, it is expected to send a data indication message with the MConReq instance to the MSAP port of

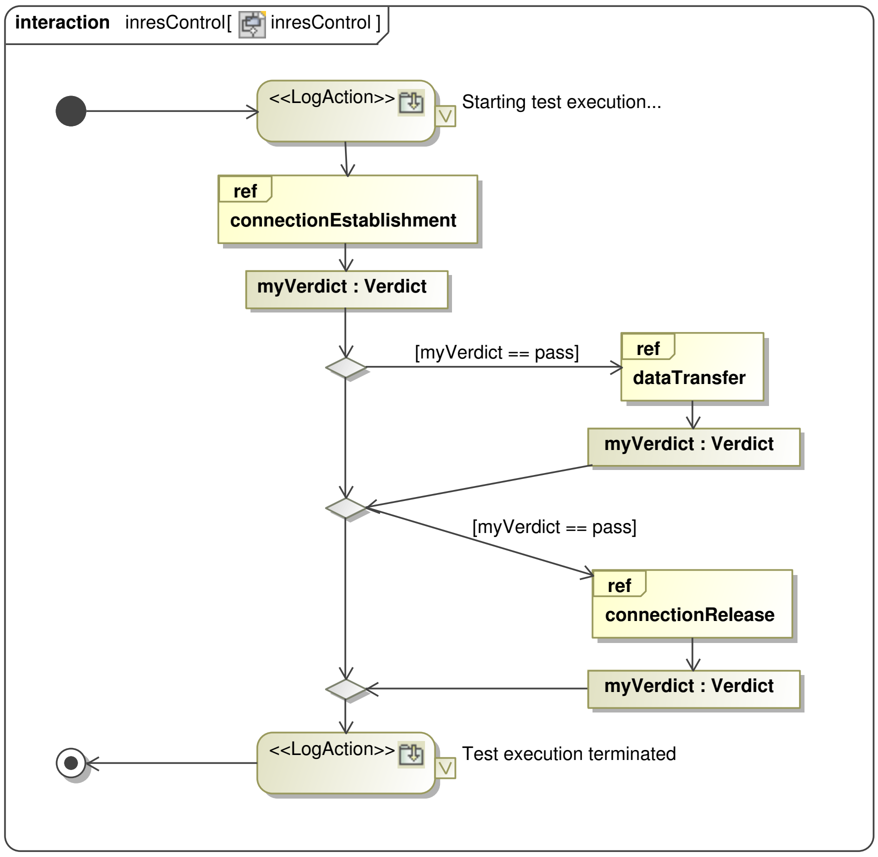


Figure 5.6: Inres test control

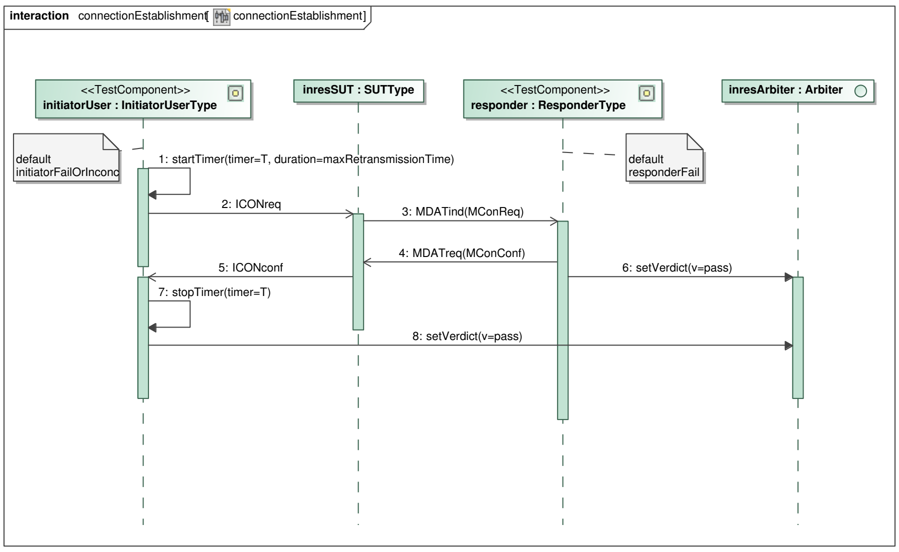


Figure 5.7: Inres test case `connectionEstablishment`

the responder. As soon as the responder gets this message, it sends back the respective connection confirmation to the SUT. After that, the responder also sets its verdict to `pass`.

Until now, the expected behavior in the test case has been described. The two defaults attached to the lifelines of the test components handle unexpected situations, which are not modeled in the interaction. The default with the name `initiatorFailOrInconc` is presented in Figure 5.8. It monitors the messages from the SUT and the running timer.

- If for some reason the SUT sends a disconnection indication, the verdict is set to `inconclusive`.
- If a message with another unexpected signal or no signal is received from the SUT, the verdict will be `fail`.
- If no message is received in a predefined time, i.e., the timer `T` generates a timeout, then the arbiter also gets the verdict `fail`.

In all three cases, the initiator user stops its behavior by finishing the test case. The second default, `responderFail`, will be executed, if any unexpected message is received by the port `MSAP` of the responder. If this happens, the verdict is set to `fail` and the responder quits its execution.

The second test case (Figure 5.9) simulates the exchange of data units. The initiator user sends out two data request messages to the `ISAP` port of the SUT. The data that is

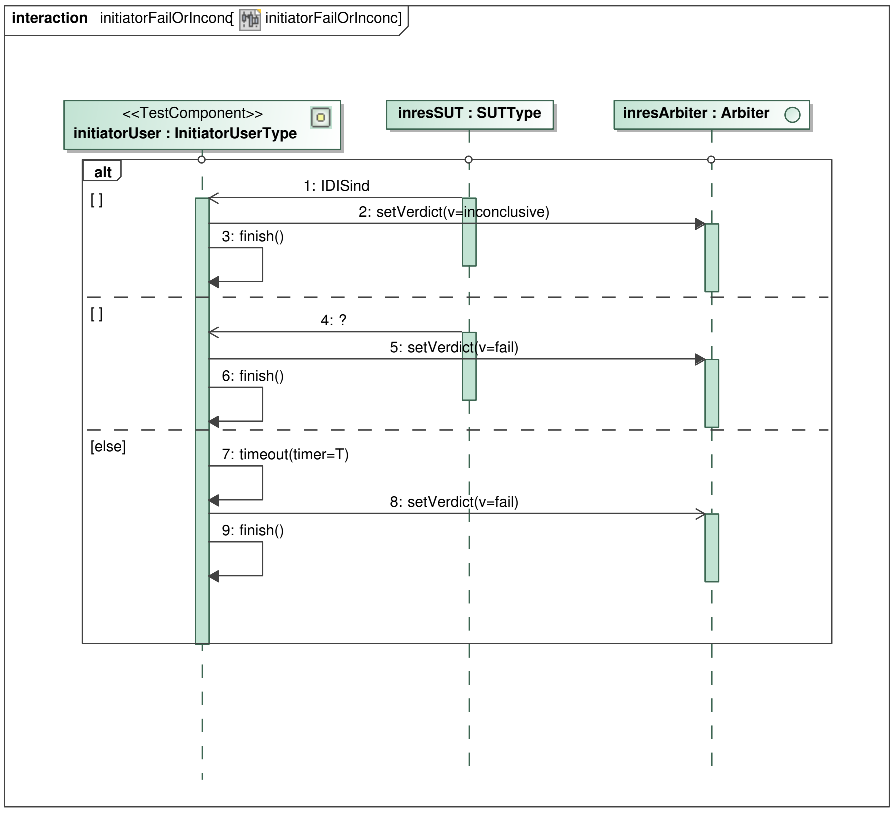


Figure 5.8: Example of a default

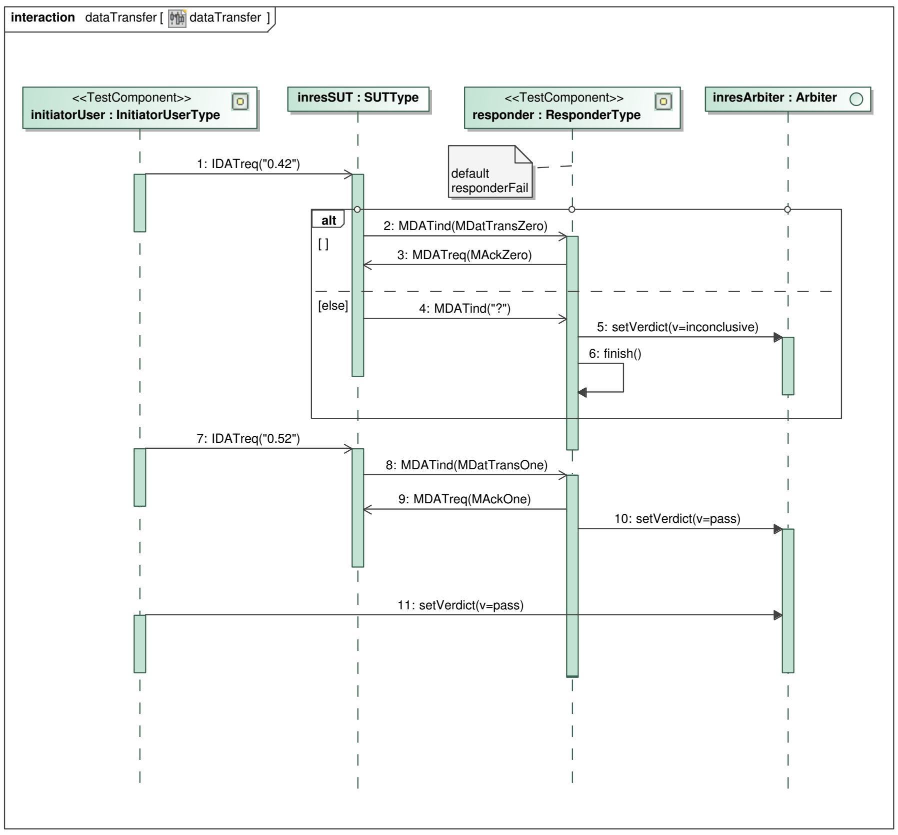


Figure 5.9: Inres data transfer test case

transmitted is just two floating-point numbers. After that, the initiator user sets its verdict to `pass`.

On receiving the first message, two alternative reactions from the SUT are possible: either it behaves as expected and sends the respective message to the responder or it sends a message with a wrong argument. The construct is an example of modeling unexpected behavior in the test case itself instead of a default. If the responder receives a data indication message with the right instance as its argument, it sends back the respective acknowledgment message and continues its execution after the `alt` block. If, however, a message with some other argument is received, the verdict of the responder is set to `inconclusive` and it finishes its execution in the test case. The second possibility might occur if the first message from the initiator user to the SUT gets lost and instead, the second one comes in its place. Additionally, any other unexpected messages are caught by the default activated for the responder.

When the SUT receives the second message from the initiator user, it is expected to forward the data unit in the instance `MDatTransOne` to the responder. If this is the case, the responder answers with the `MAckOne` instance and transfers the verdict `pass` to the arbiter.

In the last test case, the reaction of the SUT is observed while a connection is released. The responder transmits a message with the instance `MDisReq` to the port `MSAP` of the SUT. The SUT is then expected to send a disconnection request message to the initiator user. After that the two test components set their verdicts to `pass`. To catch unexpected behavior, the two defaults mentioned above are activated on the respective lifelines.

5.3 Transformation Results

The SWEUTP to SWETest transformation and the generation of TTCN-3 code described in Chapter 3 and Chapter 4 are applied to the example from the previous section. In the next two subsections, the results of the model to model and the model to text transformations are presented. Since the second model to model transformation only performs topological sortings on the test cases of an SWETest model and does not introduce any new concepts, it is not discussed here.

SWETest Model

The resulting SWETest model is stored as an XMI format. For overview reasons, exemplary sections of the model are visualized here as graphs similar to UML class diagrams. The graphs show instances of SWETest metamodel elements and the relationships between the instances. Aggregation relationships mark containment references, i.e., parent–child relations. Non-containment references are represented as directed association arrows. The

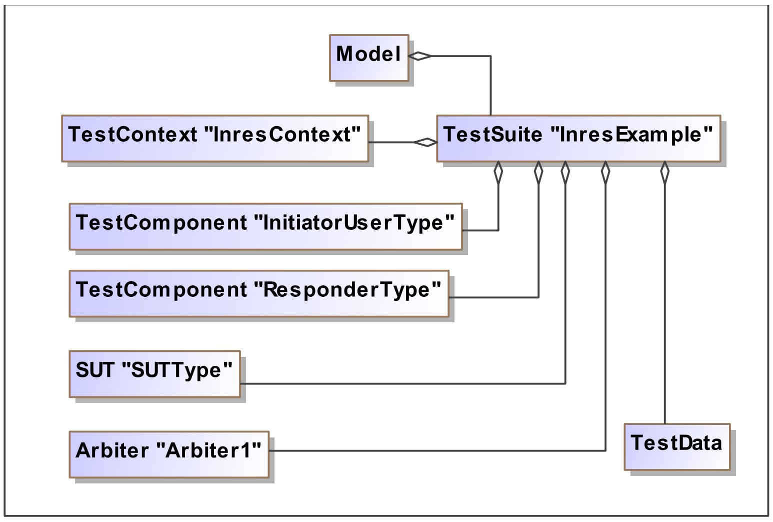

Model
TestContext "InresContext"
TestSuite "InresExample"
TestComponent "InitiatorUserType"
TestComponent "ResponderType"
SUT "SUTType"
Arbiter "Arbiter1"
TestData

Figure 5.10: Inres test suite in SWETest

value of the `name` attribute is always placed next to the type of the instance. Other attributes are listed below in a separate compartment.

The SWETest model consists of one `TestSuite` instance named `InresExample` (Figure 5.10). The test suite contains the test context, test components, and the SUT whose names correspond to the names of the classes in SWEUTP. The arbiter gets the generic name `Arbiter1`, because the predefined interface is used in SWEUTP.

Some exemplary contents of the `TestData` element are presented in Figure 5.11. Only two signals, which are referenced by port types, are shown. The attributes of the signals refer to `DataType` elements, which are contained in the data pool. The data type `UserPDU` is used as a renamed float type. The other one, named `InresPDU`, is also referenced by the only instance specification displayed here. The data type contains three attributes that define the structure of its instances. The instance specification has a `ValueSpecification` child for each of the attributes with the same names. Additionally, there are concrete values defined in the instance specification.

The type for the SUT and the test component `ResponderType` are shown in Figure 5.12. Both contain ports that refer to corresponding port types. The activity in the test component has only a finish action, because it is used to stop the behavior of the test component.

The test context is structured as displayed in Figure 5.13. The attribute `maxRetransmissionTime` is of data type `float` and contains a literal float with a value. The three interactions `connectionEstablishment`, `dataTransfer`, and `connectionRelease` represent test cases. The other two interactions store the behavior of defaults. Nodes and edges of the test control are contained in the activity `InresControl`. The test configuration consists of two connectors that reference the corresponding ports in test components and in the SUT.

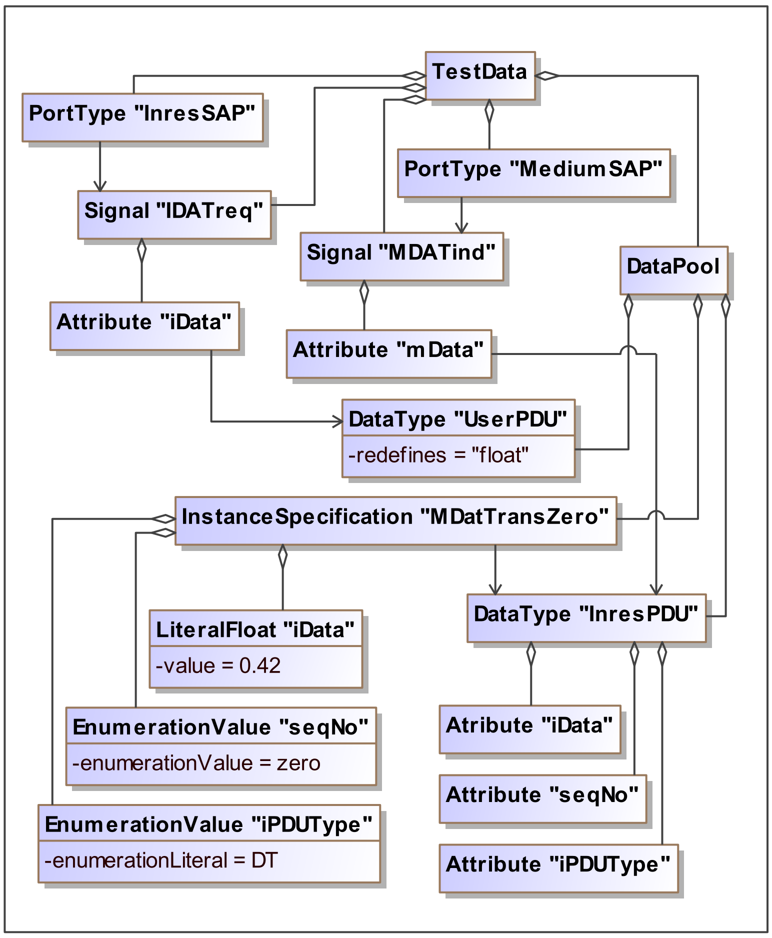


Figure 5.11: SWETest element instances representing test data

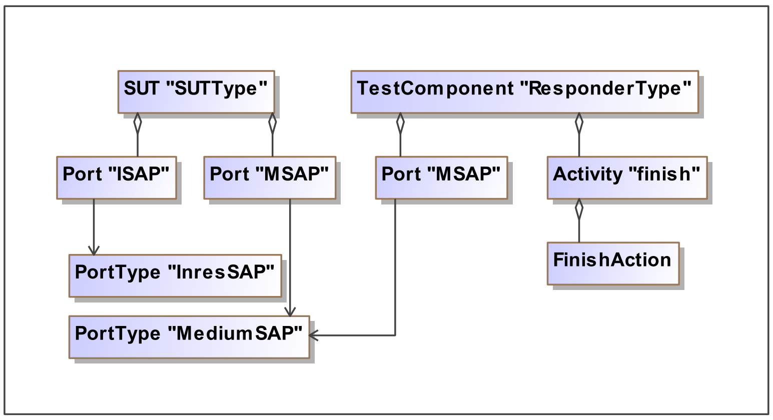


Figure 5.12: SUT and test component in SWETest

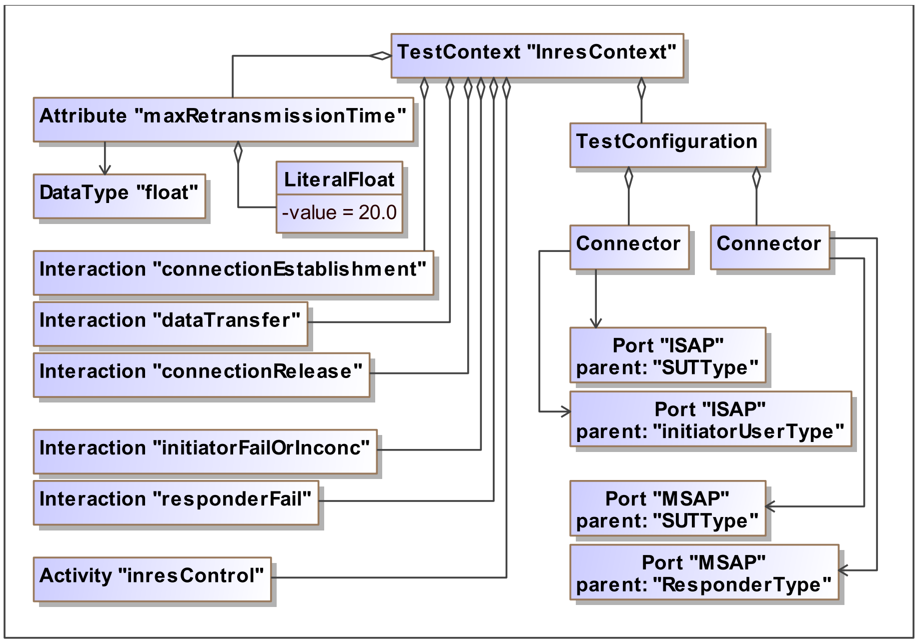

TestContext "InresContext"
Attribute "maxRetransmissionTime"
TestConfiguration
DataType "float"
LiteralFloat
-value = 20.0
Connector
Connector
Interaction "connectionEstablishment"
Interaction "dataTransfer"
Port "ISAP" parent: "SUTType"
Interaction "connectionRelease"
Port "ISAP" parent: "initiatorUserType"
Interaction "initiatorFailOrInconc"
Interaction "responderFail"
Port "MSAP" parent: "SUTType"
Activity "inresControl"
Port "MSAP" parent: "ResponderType"

Figure 5.13: Contents of the test context instance

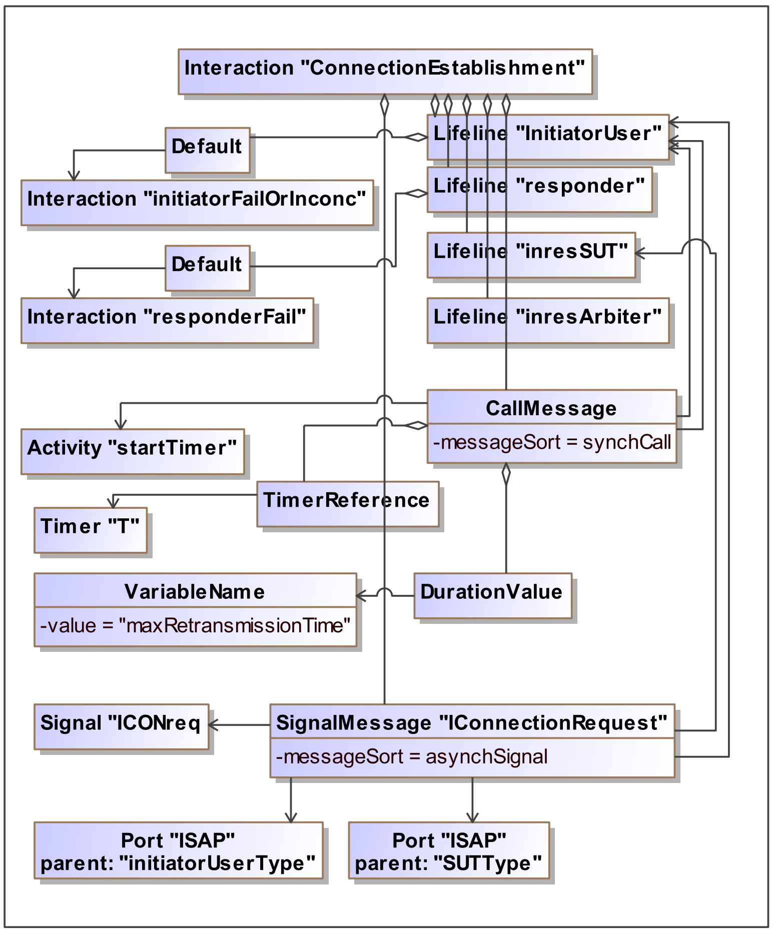


Figure 5.14: An Inres test case in SWETest

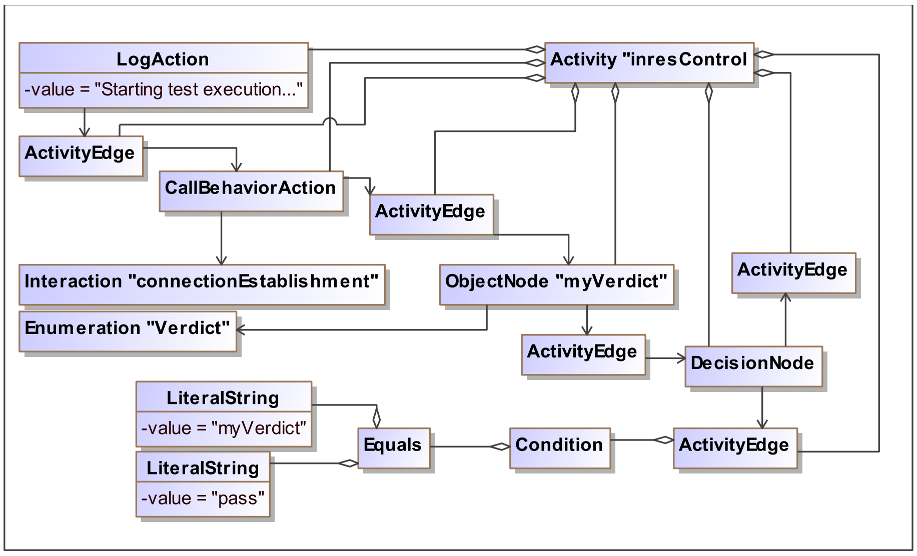


Figure 5.15: Inres test control in SWETest

As an example of an Inres test case in SWETest, the one with the name `connectionEstablishment` is shown in Figure 5.14. It contains four lifelines and a number of interaction fragments. As an example, only one call message and one signal message are displayed. The two lifelines that belong to test components have defaults, which have references to interactions with their behavior. The shown call message is sent by the `initiatorUser` to itself, which is indicated by the two references to the lifeline. The activity `startTimer`, which is supposed to be executed, is referenced in the call message. There are two arguments of types `TimerReference` and `DurationValue`. The signal message sends a signal from `initiatorUser` to `inresSUT`. The corresponding ports are referenced, as well.

in Figure 5.15, a section of the test control structure is presented. All the nodes and edges are contained in the activity `inresControl`. Starting with the log action, the activity nodes are connected by activity edges to each other. The call behavior action references the test case to be executed. The name of the object node is used in the condition to determine the path of the execution.

TTCN-3 Code

During the model to text transformation, TTCN-3 code is generated. It contains definitions for data types, components, test cases, and so on, which correspond to the Inres elements in the SWETest model. Some exemplary sections of the produced code are discussed in the following.

```
module InresExample {
  ...
}
```

The transformation generates one TTCN-3 file with a module called `InresExample`. The name is a copy of the test suite name. All the other code blocks are placed inside the module.

```
  modulepar {
    float maxRetransmissionTime := 20.0;
  }

  type float UserPDU;

  type record InresPDU {
    InresPDUType iPDUType,
    SequenceNumber seqNo optional,
    UserPDU iData optional
  };
```

For the attribute of the test context, there is a module parameter which is initialized with the `defaultValue` of the attribute. The SWETest data type `UserPDU`, which has its `redefines` property set to `float`, becomes a user-defined type definition in TTCN-3. Since `InresPDU` is a structured data type, it is a record in TTCN-3. The two fields `seqNo` and `iData` are not always set by the instance specifications of that type. Therefore, they are marked as `optional`.

```
  type record IDATreq {
    UserPDU iData
  };
  type record MDATind {
    InresPDU mData
  };

  template IDATreq IDataRequestZero := {
    iData := 0.42
  }
  template MDATind MDataTransferZero := {
    mData := {
      iPDUType := DT,
      seqNo := zero,
      iData := 0.42
    }
  }
```

All the signals, for example `IDATreq` and `MDATind` that are shown above, are transformed into records. A message that sends a certain signal becomes a TTCN-3 template of the corresponding record type. The data fields are filled with concrete data values that come from the arguments of the SWETest signal message.

```
type port MediumSAP message {
  in MDATind;
  out MDATreq;
}
type component ResponderType {
  port MediumSAP MSAP;
}
type component MainTestComponent {
}
type component SUTInterface {
  port InresSAP ISAP;
  port MediumSAP MSAP;
}
```

As an example, the TTCN-3 definitions for the port `MSAP` and for the component called `ResponderType` containing an instance of that port are presented above. The component `MainTestCommponent` is generated for the `runs on` clause of test components (see below). The SUT is represented by the component `SUTInterface`.

```
altstep initiatorFailOrInconc () runs on InitiatorUserType {
  [] ISAP.receive ( IDisconnectionIndication ) {
    setverdict ( inconc );
    stop;
  }
  [] ISAP.receive {
    setverdict ( fail );
    stop;
  }
  [] T.timeout {
    setverdict ( fail );
    stop;
  }
}
```

The behavior of a default is defined as an altstep in TTCN-3. The three possible erroneous situations are specified in blocks with `receive` statements and a `timeout` action used as conditions. The timer `T` is defined in `InitiatorUserType`. The verdicts are set with the `setverdict` directive and the behavior of the current component is finished using the `stop` statement.

```
function connectionEstablishment_initiatorUser ()
runs on InitiatorUserType {
  var default initiatorFailOrInconcDefault :=
    activate ( initiatorFailOrInconc () );
  T.start ( maxRetransmissionTime );
  ISAP.send ( IConnectionRequest );
  ISAP.receive ( IConnectionConfirmation );
  T.stop;
  setverdict ( pass );
  deactivate ( initiatorFailOrInconcDefault );
}

function connectionEstablishment_responder ()
runs on ResponderType {
  var default responderFailDefault := activate ( responderFail () );
  MSAP.receive ( MConnectionRequest );
  MSAP.send ( MConnectionConfirmation );
  setverdict ( pass );
  deactivate ( responderFailDefault );
}
```

The behavior of the two components `initiatorUser` and `responder` in the test case `connectionEstablishment` are specified in two functions, which run on the respective component types. Since there are defaults on the lifelines in the SWETest model, there are corresponding default activations and deactivations in TTCN-3. Otherwise, each code line representing behavior corresponds to an interaction fragment from the SWETest model.

```
testcase connectionEstablishment () runs on MainTestComponent
system SUTInterface {
  var InitiatorUserType initiatorUser := InitiatorUserType.create;
  var ResponderType responder := ResponderType.create;
  map ( responder:MSAP, system:MSAP );
  map ( initiatorUser:ISAP, system:ISAP );
  initiatorUser.start ( connectionEstablishment_initiatorUser () );
  responder.start ( connectionEstablishment_responder () );
}
```

The behavior of a complete test case is defined in a TTCN-3 `testcase`. First, the two components are created and stored in variables. To enable communication between the components and the interface the the SUT, their ports are then connected to each other. Finally, the behavior of the components is started referring to the functions defined previously.

```
control {
  var verdicttype myVerdict;
  log ( "Starting test execution..." );
  myVerdict := execute ( connectionEstablishment () );

  if ( myVerdict == pass ) {
    myVerdict := execute ( dataTransfer () );
  }
```

```
    if ( myVerdict == pass ) {
      myVerdict := execute ( connectionRelease () );
    }
    log ( "Test execution terminated" );
  }
```

The last section of the generated TTCN-3 code presented here defines the control part. The variable `myVerdict` is used to store the returned verdicts from the executed test cases. Only if the execution of a test case is successful, the next test case will be executed. The beginning and the finish of the control execution is logged with the `log` statement.

6 Conclusion

In this last chapter, the results of the thesis are summarized and an outlook for future work is given. In the summary, the developed method as well as the underlying implementation are referred to. Possible improvements and extensions to the method are mentioned in the outlook.

6.1 Summary

Although models are widely used in development of software systems today, the application of models in the area of software testing is still rare. The main reason for this is that there is no universally applicable method to build test models and to transform them into code. The objective of this thesis was to make a proposition for such a method.

Since UML as well as the used UML profile UTP are very complex modeling languages, a restricted subset of UTP was defined which is called SWEUTP. For the specification of the restrictions, the constraint language OCL was chosen, which is a part of the UML specification. Furthermore, a concrete syntax for conditional statements in SWEUTP models was defined. A model that conforms to SWEUTP can be processed by the transformation tool implemented in this thesis.

The method for generating test cases from models that is presented in this thesis is realized as model to model and model to text transformations. To simplify the structure of modeled test cases and to make it uniform and consistent, a new metamodel called SWETest was created, which is based on Ecore. SWETest is used as the output format in the model to model transformation from SWEUTP. The transformation process was implemented using the language Xtend which is part of openArchitectureWare.

SWETest models containing test cases can easily be altered by further model to model transformations. The example that is given in this thesis changes SWETest models in the way that all the distributed test components are merged into one test component and deterministic test behavior is created with the help of a topological sorting algorithm. The process was implemented in Xtend as an in-place transformation, i.e., the model is changed directly and no new model is generated.

The last transformation method presented is a model to text transformation that generates code for test cases. The input is an SWETest model containing one or more test suites. Each test suite is mapped to a TTCN-3 module consisting of all the definitions that are needed to execute the test cases. This transformation is implemented in Xpand, a template oriented language which is also part of oAW.

6.2 Outlook

The presented method can be regarded as an approach to Model Driven Testing as described by Baker et al.[17] Just as Baker et al. suggest, the test data is contained in the UTP model along with the test behavior and the test architecture. However, data for testing a system is likely to be very extensive, since even small systems have a large number of possible input and output values. Modeling the test data can be very time-consuming. Therefore, it should be considered to create a possibility to define the test data externally and import it during the transformation. For example, the data could be imported from a database or an XML document.

The defined UTP variant called SWEUTP does not contain all the concepts of UTP. For example, data selectors, schedulers, and timezones are not processed by the transformations. Consequently, there are no corresponding elements in the SWETest metamodel and no mappings to TTCN-3. Ideally, all the UTP elements should be included in SWEUTP and the transformation algorithms should be adapted to process them. The mappings to TTCN-3 can be implemented as proposed in the UTP specification [33]. For example, data selectors could be represented as external functions.

During the generation of TTCN-3 test cases, only the combined fragments with the operators `alt`, `opt`, and `loop` are processed. Additional mappings for other kinds should be defined and implemented to allow more complex structures for the behavior of the test cases. The operator `par`, for example, could be used to explicitly mark parallel regions of the test behavior. Invalid behavior sections that lead to unsuccessful verdicts might be represented in `neg` sections.

So far, only message-based communication between the test components and the SUT was considered. In SWEUTP, operation calls are only sent onto the same lifeline and to the arbiter. It is also possible to add corresponding model elements and transformation rules for procedure-based communication between the test components and the SUT. The transformation to TTCN-3 would be straightforward, since the language contains corresponding concepts.

Finally, generation of code for other programming languages other than TTCN-3 can be implemented. For example, test cases could be generated for the Java-based framework JUnit [8] or the multi-purpose programming language Ruby [13].

Appendix A: UML metamodel

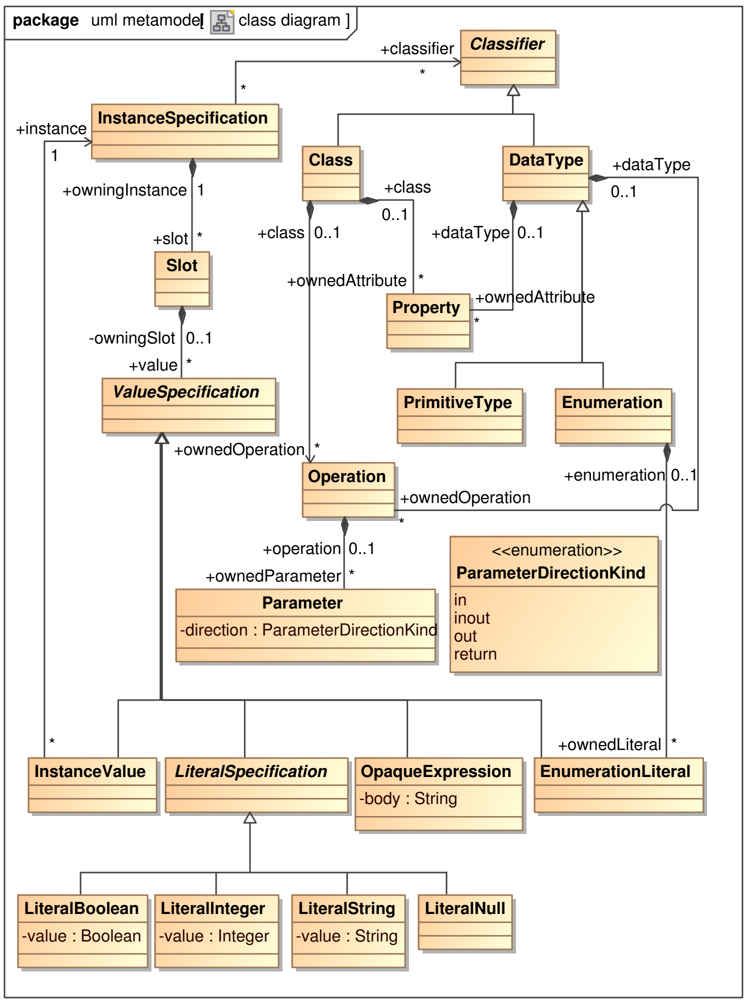


Figure 6.1: Part of the UML metamodel for class diagrams

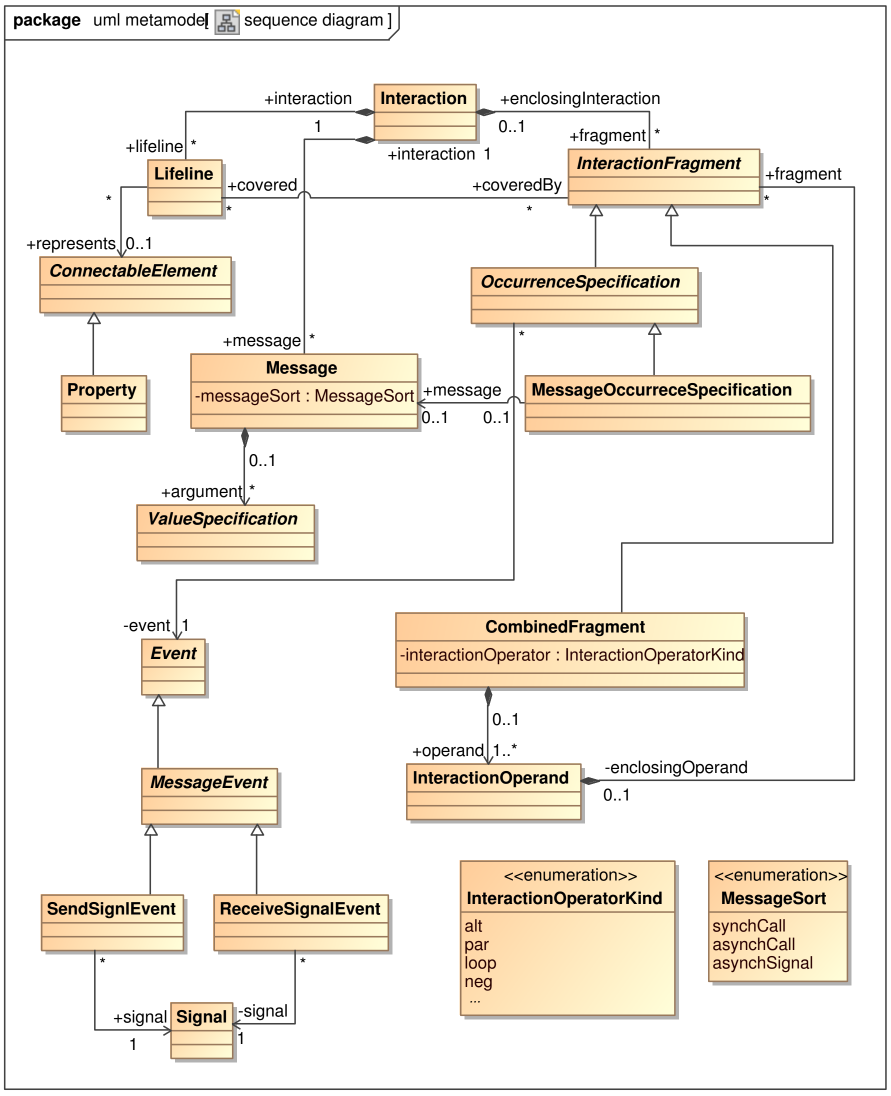


Figure 6.2: Metaclasses defining UML sequence diagrams

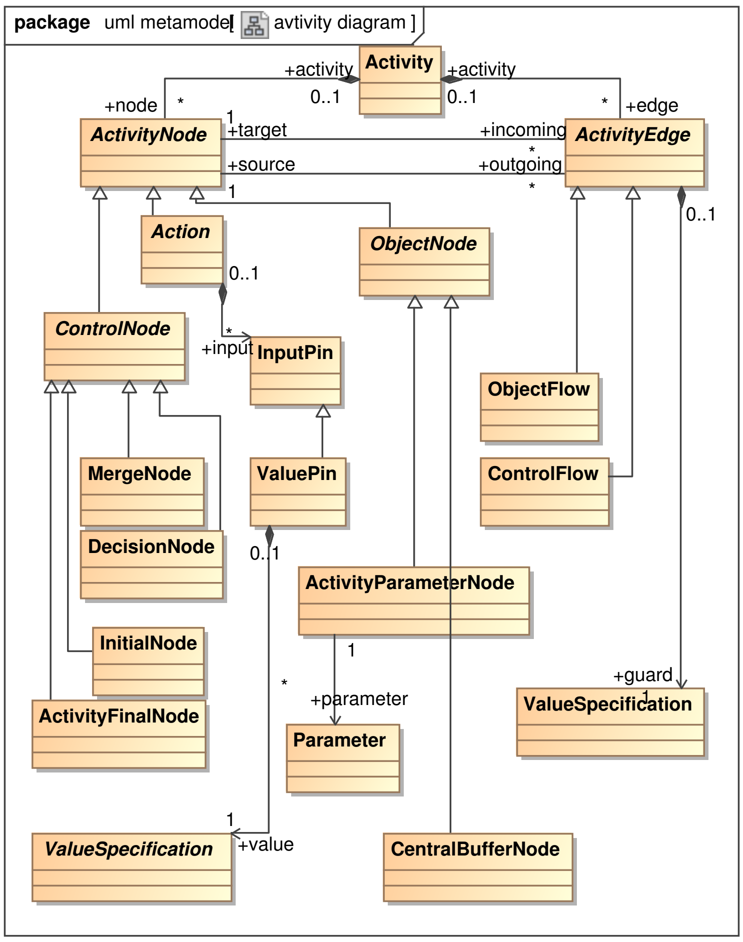


Figure 6.3: Metamodel for activity diagrams

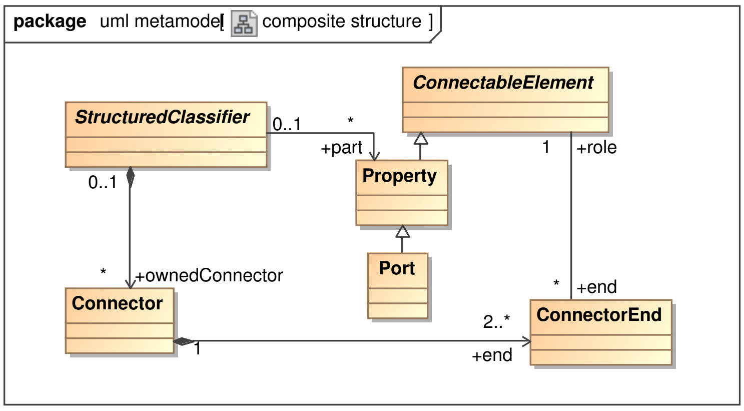


Figure 6.4: Metamodel for composite structure diagrams

Appendix B: Modeling Test Cases with Magic Draw

Step-by-step procedure for modeling a test suite in Magic Draw.

1. Basic structure of the test suite:
    - Create a new project using the UTP template.
    - Create a package which will contain all the test elements and test cases.
    - Create a Test Architecture Diagram in the package.
    - Create a Test Context element in that diagram.
    - Create a Classifier Behavior in the Test Context. Go to Specification, select Activity next to Classifier Behavior, type in a name for the behavior. This behavior will correspond to the control part in the TTCN-3 code.
    - Create a Test Overview Diagram inside the behavior: Right-click on the behavior in the Containment Tree and select 'New Diagram'. The Test Overview Diagram is part of this work, so it has to be installed manually.
    - Similarly, create a Test Configuration Diagram in the Test Context.
    - Create a number of Sequence Diagrams in the Test Context. Those will correspond to the test cases which will be generated in TTCN-3.
    - Optionally create a package for test data (like Signals, Datatypes and Instances). The advantage of this is that this package can be shared between several test suites.
    - In this package, create a Test Architecture Diagram on which all the test data will be placed.
2. Active Test Elements and connections between them:
    - Create Test Components and an Arbiter in the Test Architecture Diagram with the TestContext. The Arbiter will be responsible for setting the Verdict.
    - Create a simple Class in the Test Architecture Diagram, which will be used as the type for the SUT element.
    - In the Test Configuration Diagram, create a SUT Part and Test Component Parts. Choose the previously created Test Components and the SUT Class as types for

the Parts. Type in the names for the Parts. Note that corresponding attributes are added automatically to the Test Context.

- On the Test Architecture Diagram for the test data, create Classes, which will serve as types for Ports.
- In the Test Configuration Diagram, add Ports to the Test Components and the SUT. Select the previously created Classes as types. The ports of the SUT will be the 'system:'-ports in TTCN-3.
- Connect the Ports, which should be mapped to each other, with Connectors.

3. Data to be sent in Messages:
    - In the Test Architecture Diagram in the Data Package, create Signals, which should be sent between Test Components and the SUT. The Signals can have Attributes of primitive types (like Integer or String) or of user-defined types (see later). The Signals will be transformed to records in TTCN-3.
    - Draw Directed Association arrows from Port types to the Signals, which should be sent through the Ports.
    - Define your own Data Types and Enumerations. If a Data Type should be derived from a primitive type like String or float, the primitive type must be given in the BaseClassifier of the Data Type. Like Signals, own DataTypes can have attributes, in which case they are also transformed to records. If EnumerationLiterals are supposed to have other integer values than 0, 1, etc., then each value can be specified in the Specification of the EnumerationLiterals, in the field 'Specification'.
    - Create Instances of your own structured Data Types. Those can be sent as arguments in Signals later in the Sequence Diagrams. The Instances are transformed to (inner) templates in TTCN-3 (as fields inside of 'Signal'-templates, see later).
    - Instead of regular values, some special values can be defined in the Slots of the Instances:
        - A LiteralNull results in an omit-statement.
        - A LiteralString stereotyped as LiteralAnyOrNull results in a *.
        - A LiteralString with the value 'parameter' and stereotyped as LiteralAny generates a parameter for the template.
        - A LiteralString with any other value stereotyped as LiteralAny results in a ?

4. Basic test cases
    - Add Operations to the TestContext. Assign the TestCase stereotype to them if it is not done automatically.

- Connect the Operations to the Interactions by choosing the right ones for the Method field in the Specification dialog.
- Create the Arbiter which will accept and set Verdicts:
  - Add an attribute to the TestContext with the predefined type Arbiter (interface) or
  - Create your own Arbiter Class which implements the Arbiter interface and add an attribute of that type to the TestContext.
- In the Interactions, create Lifelines out of the attributes of the TestContext (TestComponents, SUT, Arbiter).
- Following elements can be created on the Lifelines:
  - 'Call Message' to the Arbiter to set or get the verdict. Choose the right operation in the Operation field. MagicDraw might generate a string attribute for the setVerdict operation. Delete it and create a new attribute of type InstanceValue. Choose a verdict value from the enumeration Verdict in the UTP package.
  - 'Send Message' to send Signals between the TestComponents and the SUT. The name for the message must be given (it is used to generate a template in TTCN-3). The arguments of the message must correspond to the types of the Signal attributes or can be wildcards (as described for Instance Slots above). Different messages may have the same name if they contain the same signal with the same arguments (in this case only one template will be generated). The ports can be specified by choosing a Connector. If the Connector field is empty, then an 'any port' statement will be generated in TTCN-3. The Signal does not have to be given in messages that come from the SUT. In that case a 'receive' statement with no arguments will be generated.
  - 'Message to self' to call an Activity (see below) or a nativeCode operation. An operation with the name 'nativeCode' can be used to input a code line of the language that will be generated. This is useful if some features are needed that are not (yet) implemented in the transformation process.

5. Activities and timers
   - Create Activities in the TestComponents to be able to use some UTP Actions. Activities with the following Actions inside them can be created:
     - StartTimerAction
     - StopTimerAction
     - TimeOutAction
     - FinishAction

- Create a corresponding operation for each Activity in the TestComponents and assign the Activity to the operation:
  - The operation that starts a timer must have two parameters: one of type Timer and one of type Duration (from the UTP package).
  - The one that stops a timer must have a Timer parameter.
  - The timeout operation also must have a Timer parameter.
- Assign the Activities to the corresponding operations. For each parameter of the operation, one activity parameter and one parameter node are generated. Give names to all the activity parameters.
- The generated parameter nodes can be dragged onto the Activity Diagram and used as input for the Action.
- Add attributes of type Timer to the TestComponents to be able to use timers in the test cases.
- To add an execution of a UTP Action to a lifeline, create a Message to self, choose the desired operation (which is assigned to the Activity with the Action) and specify the arguments. For a timer, the argument must be the name of a Timer. Duration can be a numerical value or a variable name (see below). The arguments must be OpaqueExpressions, since MagicDraw's 'ElementValue' is not in the UML standard and thus will not be exported to the EMF UML file. The simplest way to create arguments as OpaqueExpressions is to just type in the values directly in the Sequence Diagram.
- Attributes of the TestContext which are not TestComponents, SUT, or Arbiters are treated like 'global variables' for the test cases. Their names can be used as arguments in the Call Messages/Messages to self. The initial values for those attributes can be typed in in the TestContext definition in the Diagram. In TTCN-3, modulepar parameters are generated.

6. Defaults
   - Create a new Sequence Diagram in the TestContext which will be a default.
   - A default is properly generated in TTCN-3, if it consists of one alt block.
   - Assign the default Interaction to a test component: Create a Comment and attach it to a lifeline of a test component. The text of the comment must start with 'default<enter>' and then contain the name of the Interaction.

7. Test control
   - In the Activity diagram for the classifier behavior of the TestContext, create an initial node and an activity final node.
   - Create LogActions to log some statements. Create a ValuePin in each LogAction and type in the statement that should be logged in the 'value' field. MagicDraw

shows an error message if a pin is created from the toolbar on the LogAction, so create the pin in the Specification dialog of the LogAction (either under Request or under Target).

- Create InteractionUses and choose Interactions.
- An ObjectNode following an InteractionUse means that the result of the test case (the verdict) is stored in the object.
- An object can be passed to a decision node. Outgoing edges of the decision node should have guards to specify conditions. An outgoing edge without a guard indicates the main flow. Different flows can be merged with merge nodes.

Acknowledgments

First of all, I would like to thank my tutor Benjamin Zeiss who always had an answer to my questions and always responded to my countless e-mails, no matter at what time of day (or night). Also, a big thank to Prof. Dr. Jens Grabowski who made this thesis possible. A special thank to my family and friends for their great support.

List of Figures

Abbreviations and Acronyms

EMF	Eclipse Modeling Framework
ETSI	European Telecommunications Standards Institute
MBT	Model Based Testing
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MDSD	Model Driven Software Development
MDT	Model Driven Testing
MOF	Meta-Object Facility
MSC	Message Sequence Chart
OCL	Object Constraint Language
oAW	openArchitectureWare
OMG	Object Management Group
SAMSTAG	Sdl And Msc baSed Test cAse Generation
SDL	Specification and Description Language
SUT	System Under Test
TTCN-3	Testing and Test Control Notation Version 3
UML	Unified Modeling Language
UTP	UML Testing Profile
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Bibliography

[1] ANTLR Parser Generator. http://www.antlr.org/. [Accessed October 22, 2009].

[2] Apache JMeter. http://jakarta.apache.org/jmeter/. [Accessed October 28, 2009].

[3] Conformiq – automated test design. http://www.conformiq.com/products.php. [Accessed October 31, 2009].

[4] Eclipse Modeling - MDT Home. http://www.eclipse.org/modeling/mdt/. [Accessed September 14, 2009].

[5] Eclipse.org home. http://www.eclipse.org/. [Accessed August 2, 2009].

[6] Elvior – software testing professionals. http://www.elvior.ee/motes/. [Accessed October 31, 2009].

[7] ETSI. http://www.etsi.org. [Accessed September 9, 2009].

[8] JUnit.org Resources for Test Driven Development. http://junit.org/. [Accessed October 31, 2009].

[9] MagicDraw - Architecture made simple. http://www.magicdraw.com. [Accessed August 27, 2009].

[10] Model Driven Test Development. http://www.mdtd.de/. [Accessed October 28, 2009].

[11] Object Management Group. http://www.omg.org. [Accessed Juli 22, 2009].

[12] Official openArchitectureWare Homepage. http://www.openarchitectureware.org/. [Accessed August 2, 2009].

[13] Ruby Programming Language. http://www.ruby-lang.org/de/. [Accessed October 31, 2009].

[14] Tefkat – The EMF Transformation Engine. http://tefkat.sourceforge.net/. [Accessed October 28, 2009].

[15] The test sequence generator TGV. `http://www-verimag.imag.fr/~async/TGV/index.shtml.en`. [Accessed October 31, 2009].

[16] TTCN-3 home page. `http://www.ttcn-3.org`. [Accessed September 9, 2009].

[17] *Model-Driven Testing Using the UML Testing Profile*. Springer Berlin Heidelberg, October 2007.

[18] L. Apfelbaum and J. Doyle. Model based testing. In *Software Quality Week Conference*, 1997.

[19] J. Bezivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, May 2005.

[20] E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science (LNCS)*, pages 187–195. Springer, 2001.

[21] Z. R. Dai. *An Approach to Model-Driven Testing: Functional and Real-Time Testing with UML 2.0, U2TP and TTCN-3*. PhD thesis, November 2006.

[22] J. M. Favre. Towards a basic theory to model – model driven engineering. In *Proceedings of Workshop on Software Model Engineering, WISME 2004*.

[23] D. Gasevic, D. Djuric, and V. Devedzic. *Model Driven Engeneering and Ontology Development*. Springer, second edition, June 2009.

[24] J. Grabowski. Sdl and msc based test case generation – an overall view of the samstag method. Technical report, University of Berne, Institute for Informatics, May 1994.

[25] D. Hogrefe. OSI formal specification case study: The inres protocol and service. Technical report, University of Berne, Institute for Informatics and Applied Mathematics, May 1991.

[26] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall International, November 1990.

[27] International Telecommunication Union Telecommunication Standardization Sector (ITU-T). *Message Sequence Chart (MSC)*, November 1999. Available online at `http://www.itu.int/ITU-T/studygroups/com10/languages/Z.120_1199.pdf`.

[28] International Telecommunication Union Telecommunication Standardization Sector (ITU-T). *Specification and description language (SDL)*, November 1999. Available online at `http://www.itu.int/ITU-T/studygroups/com10/languages/Z.100_1199.pdf`.

[29] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, November 1962.

[30] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc., second edition, July 2004.

[31] H. W. Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, University of Goettingen, 2004.

[32] Object Management Group (OMG). *MDA Guide Version 1.0.1*, June 2003. Available online at `http://www.omg.org/docs/formal/03-06-01.pdf`.

[33] Object Management Group (OMG). *UML Testing Profile, Version 1.0*, July 2005. Available online at `http://www.omg.org/docs/formal/05-07-07.pdf`.

[34] Object Management Group (OMG). *Meta Object Facility (MOF) Core Specification*, January 2006. Available online at `http://www.omg.org/docs/formal/06-01-01.pdf`.

[35] Object Management Group (OMG). *Object Constraint Language*, May 2006. Available online at `http://www.omg.org/docs/formal/06-05-01.pdf`.

[36] Object Management Group (OMG). *MOF 2.0/XMI Mapping, Version 2.1.1*, December 2007. Available online at `http://www.omg.org/docs/formal/07-12-02.pdf`.

[37] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*, November 2007. Available online at `http://www.omg.org/docs/formal/07-11-04.pdf`.

[38] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*, November 2007. Available online at `http://www.omg.org/docs/formal/07-11-02.pdf`.

[39] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.0*, April 2008. Available online at `http://www.omg.org/docs/formal/08-04-03.pdf`.

[40] G. Pruesse and F. Ruskey. Generating linear extensions fast. *SIAM Journal on Computing*, 23(2):373–386, April 1994.

[41] D. P. Sidhu and T.-K. Leung. Formal Methods for Protocol Testing: A Detailed Study. *IEEE Transactions on Software Engineering*, 15(4):413–426, April 1989.

[42] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, second edition, February 2009.

[43] M. Utting and B. Legeard. *Practical Model-Based Testing. A Tools Approach.* Morgan Kaufmann, March 2007.

[44] Y. L. Varol and D. Rotem. An algorithm to generate all topological sorting arrangements. *The Computer Journal*, 24(1):83–84, 1981.

[45] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din. From U2TP models to executable tests with TTCN-3 – an approach to model driven testing. In *Testing of Communicating Systems*, pages 289–303, 2005.