An Approach to Model-Driven Testing

– Functional and Real-Time Testing with *UML 2.0*, *U2TP* and *TTCN-3*

Technische Universität Berlin

Zur Erlangung des akademischen Grades der
Doktorin der Ingenieurwissenschaften

– ***Dr.-Ing.*** –

vorgelegt von

Dipl.-Inf. Zhen Ru Dai

6. Juni 2006

Abstract

The Model-Driven Architecture (MDA) aims at better software engineering by enhancing the productivity, portability and interoperability of software. To guarantee the quality of the resulting software systems, testing plays a decisive role.

In this work, an approach to model-driven testing based on the idea of the MDA framework is presented. It introduces a new test modeling language and a test generation technique in order to develop both functional and real-time tests by models. The former is defined as an extension of a standardized system modeling language to facilitate test model generation from system models. The latter presumes the availability of a system model which can be reused for test specification - a prerequisite that is almost always true in modern software development processes. Through different abstraction levels, the test model can be adapted to platforms on which test code is to run.

As a result, test code that are executable on different platforms can be generated from a single model. The effectiveness of this approach is demonstrated by testing a Bluetooth application.

Zusammenfassung

Die Model-Driven Architecture (MDA) hat es sich zum Ziel gesetzt, Software bezüglich ihrer Produktivität, Portabilität und Interoperabilität zu verbessern. Für die Qualitätssicherung dieser Software spielt Testen eine entscheidende Rolle.

In der vorliegenden Arbeit wird ein Ansatz zum modell-getriebenen Testen vorgestellt, der auf den Ideen von MDA aufbaut. Der Ansatz stellt eine neue Test-Modellierungssprache und eine Test-Generierungstechnik vor, die der funktionalen sowie Realzeit-Testentwicklung aus Modellen dienen. Die Test-Modellierungssprache ist als Erweiterung einer System-Modellierungssprache definiert, um eine Ableitung von Test-Modellen aus System-Modellen zu ermöglichen. Die Test-Generierungstechnik geht davon aus, dass ein System-Modell existiert, welches für die Testspezifikation wieder verwendet werden kann – eine Voraussetzung, welche in den modernen Softwareentwicklungsprozessen fast immer erfüllt ist. Über mehrere Abstraktionsschritte wird das Test-Modell auf Plattformen angepasst, auf denen später der Testkode ausgeführt wird.

Mit diesem Ansatz können somit aus einem einzigen Model Testkodes für unterschiedliche Plattformen erzeugt werden. Die Anwendbarkeit dieser neuen Test-Modellierungssprache und Test-Generierungstechnik wird anhand einer Fallstudie, bei der eine Bluetooth-Applikation getestet wird, demonstriert.

Acknowledgements

This work has been developed in close cooperation with the University of Lübeck and Fraunhofer FOKUS.

Of all the people, I owe the most to Professor Dr. Ina Schieferdecker and Professor Dr. Jens Grabowski who advised my work during the past few years. Their knowledge and suggestions have proven invaluable to me and contributed much to the results presented in this thesis. For all their efforts, I would like to express my heartiest gratitude to both of them.

I also would like to thank my former supervisor Professor Dr. Dieter Hogrefe both for his professional advices and for providing me with excellent working conditions during my stay as research assistant at the University of Lübeck.

My thanks are also tendered to the other U2TP consortium members: Paul Baker, Oystein Haugen, Serge Lucio, Eric Samuelson and Clay Williams. I had the honour discussing with those skilled people. It was fun working with you guys and I am looking forward to continuing our collaboration!

Furthermore, I am grateful to all my former colleagues in Lübeck and current colleagues at Fraunhofer FOKUS in Berlin. The discussions I shared with them were extremely fruitful. In particular, I truly appreciate the tireless efforts and proof-reading of Helmut Neukirchen. Also, special thanks are dedicated to my FOKUS' colleagues for their encouragement, especially to George Din, Andreas Hoffmann, Axel Rennoch and Justyna Zander-Nowicka. I also wish to acknowledge the great interest of Keith Duddy in my model transformation. This thesis would not be in its current shape without the endeavor of all these people.

Last but not least, I would like to thank my parents. All this would not be possible without your endless love, encouragement and faith in me. You gave me the best education a child can wish. I will always be proud of being your child... come what may!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"The beginning is the most important part of the work." – *Plato*

With the growing complexity of software systems, solid testing becomes more and more important in order to guarantee the quality of software. Models lift up the abstraction levels for coding and increase the ***reusability*** and ***productivity*** during the software development process. Using models for test ***automation***, test code can be generated ***efficiently***. In order to specify ***extensive*** tests, a close ***collaboration*** between system development and test development is indispensible. In the traditional software development processes such as V-Model [DW99] and its two modifications called V-Model eXtreme Tailoring (V-Model XT) [V-M04] and W-Model [SL03] or Rational Unified Process (RUP) [Kru00], test development steps are strongly coupled with system development steps. Since several years, it can be observed that many research works are done on a new framework for software development process called the Model-Driven Architecture (MDA)[1].

MDA addresses the increasing interoperability problem during the software development process caused by the various platforms where software runs on, such as CORBA or .NET [OMG03]. The key of MDA is a model which is specified independently from its middleware platform. By means of transformation and refinement techniques, platform-specific information can be automatically added to the model. Subsequently, system code running on diverse platforms can be generated from models. As a consequence, when following the MDA Framework, system code running on different platforms can be generated from a single model.

Further benefits of MDA are that it sets upon standardized languages and technologies to enable e.g. better tooling. Standards under the umbrella of MDA comprehend the Unified Modeling Language (UML), the Meta-Object Facility (MOF), the XML Metadata Interchange (XMI) and the Queries/-Views/Transformations (QVT).

[1] The MDA Framework has been introduced by the Object Management Group (OMG).

Indeed, automatic transformations and refinements provide important technologies for code generation within MDA. Nevertheless, failures may creep in between the different abstraction levels of the models. Especially when adding platform-specific information to a model, failures on configuration or interfaces may lead to incorrect system code. These failures need to be tested. Model-based test generation has been a challenge for decades [BJK+05]. But most of these approaches only generate test code for a single platform. Anyhow, specifying tests for software running on different platforms turns out to be more laborious than writing test code running on a single platform, simply because for each platform, a test has to be specified. Furthermore, software system and test system do not have to run on the same platform either. This makes test specification all the more complex.

While much work has been done to utilize MDA for system development [FP04, Bez01, KCL03], MDA does not address test within its process at all! In this PhD thesis, research work has been done on test development based on the ideas of MDA. By means of this test approach, test code running on different platforms can be generated automatically from a single model.

The problem this PhD thesis adresses can be stated as follows:

Can we use a model-driven approach for test development, similar to that of MDA to improve software quality? If true, what problems will we be confronted with?

In order to align with the terminology of MDA, we call the test approach the Model-Driven Testing (MDT). However, before having a deeper insight into the MDT approach, the following questions should be examined more closely:

- Which prerequisites must a modeling language fulfill in order to specify tests? Moreover, is it possible to use a common language for both system and test specifications?
- How can models be reused in an efficient way to derive tests? Which criteria must these models meet?
- Can test requirements be explicitly incorporated into a test model during test automation?
- How can test models be carried into execution for both functional and real-time tests?

In order to realize the MDT approach, the thesis considers the following main aspects:

Choosing UML as modeling language Models are the main artefacts within the MDT development process. Therefore, it is important to have a well-defined test modeling language. A common modeling language for both system and test specification enhances and optimizes the collaboration work between system and test developers. Although various modeling languages exist [IT96, Z1299, GV03, CKLY98, Sif89], the MDA Framework recommends UML [JBR99] as the appealing and sophisticated modeling language. Hence, in this work, the newest version of the Unified Modeling Language, version 2 (UML 2.0) is examined in detail for test specification. As a result, a new ***test modeling language*** extending UML 2.0 is defined and standardized where the author of this document actively participated to its definition in the context of this thesis.

Using transformation techniques for test generation In order to bridge the gap between system and test development, a test model shall be generated from an existing system model. Based on transformation techniques defined in the MDA Framework, ***formal transformation rules*** are defined in this work for the automatic generation of test models. These rules are implemented in a well-defined model transformer language.

Modeling test requirements Test requirements can be mostly deduced from system requirements, such as describing the relationship between a triggering and its responding event. However, there are also test-specific requirements which do not exist in system requirements but are to be considered along automatic test generation. In order to assure test quality, test requirements shall be explicitely integrated during test generation. Following this issue, we defined a MOF-based[2] ***meta-model for test requirement specification***.

Generating executable code for functional and real-time tests Most of the existing test languages can be used to specify functional tests. Beside this, real-time tests become more and more important. Herein, the evaluation does not only depend on the correct behavior of the system under test, but also proves whether a system meets the time restrictions defined in its requirements, e.g. delay or jitter. The Testing and Test Control Notation, version 3 (TTCN-3) is a standardized testing language which we utilize in our MDT approach for test execution. In the current TTCN-3 standard, real-time test aspects are not fully covered. Consequently, TTCN-3 is augmented with ***real-time test concepts*** in this work.

[2]To be precise, the meta-model is based on the Meta-Object Facility, version 2 (MOF 2.0) standard.

The thesis is organized as follows: This chapter gives an overview of the research topic of this PhD thesis. It introduces the problems the work is dealing with, its objectives and main assumptions to its reader.

Chapter 2 discusses the fundamental principles of testing and the MDA Framework. A survey of standardized languages and technologies gives ideas of how the MDA Framework may be enhanced for testing.

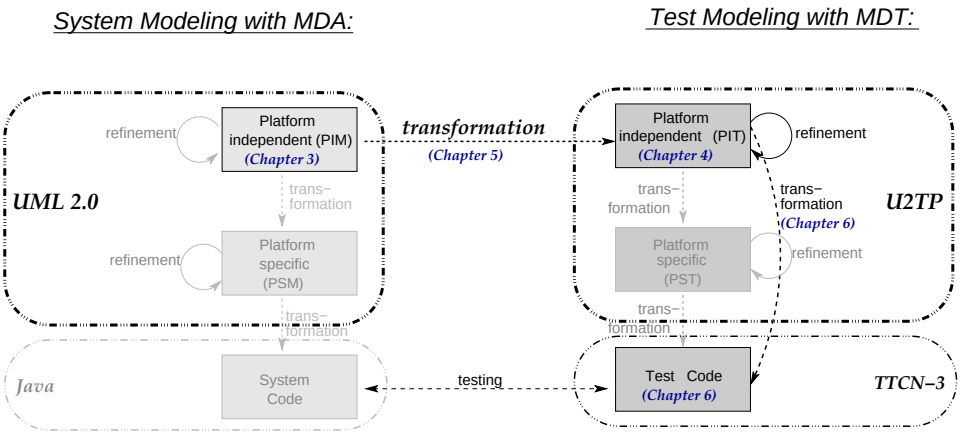


Figure 1.1: Chapter Overview

Figure 1.1 shows a road-map of the successive chapters. Chapter 3 introduces system modeling with UML 2.0 applied to a Bluetooth application as a case study. This case study will accompany the reader throughout the whole thesis. The Bluetooth system is modeled platform-independently by various UML 2.0 diagrams. This system model serves as input model for test model generation in the succeeding chapters.

Chapter 4 analyzes the relevance of UML 2.0 for test modeling. As a result, an extension of UML 2.0 called the UML 2.0 Testing Profile (U2TP) is needed. This profile has been defined at the OMG where the author of this work is an active member of the U2TP consortium. Meanwhile, U2TP has become an official standard at the OMG. In this chapter, the concepts of U2TP and the motives of their definition are discussed in detail. Its usage is illustrated by defining a U2TP test model for the Bluetooth application based on the system model presented in Chapter 3.

Chapter 5 deals with another important issue: It shows how a system model may be reused efficiently for test specification. Here, we show that various diagrams defined in the system model can be adopted to a test model[3].

[3]Although MDA introduces an even more abstract model level called Computation

Still, this test model is just a skeleton of the final test model and various refinements must be done. As a result, a guideline about model reuse and refinement is provided. Based on the guideline, a U2TP model can be automatically generated. This is realized by well-defined transformation rules implemented by a formal transformation language called *Tefkat*. Furthermore, we incorporate a *Test Directive Meta-Model* into our transformation rules to enable explicit test requirement integration during model transformation.

Chapter 6 deals with execution of functional and real-time tests. It examines TTCN-3 test code generation from existing U2TP test models. The necessity of extending TTCN-3 for real-time test specification is explained and new real-time test concepts are defined. Mapping rules between both language concepts are defined and test code for the Bluetooth case study is presented applying the mapping rules to the test model. Eventually, test code generated from the test model can be run against the system code. Last but not least, we complete the work with a summary and some concluding words.

Independent Model (CIM) for domain model specification [OMG03], it is much too vague for our model transformation approach and is thus out of our scope.

Chapter 2

Fundamentals on MDA and Testing

"The sooner you start, the longer it takes." – *Mythical Man Month*

In this chapter, basics about the MDA Framework and testing is presented. Section 2.1 gives an insight into the model-driven system development approach provided by the OMG. The underlying technologies of MDA are discussed in detail. Furthermore, an existing transformation language called *Tefkat* is outlined. In Section 2.2, test development phases and important testing aspects are described by categorizing those in dimensions. At last, the established test language TTCN-3 is introduced.

2.1 The MDA Framework

Nowadays, the complexity of modern systems can no longer be managed at the code level. Model-based automation for system development has become an important issue. Furthermore, systems are never built using only one implementation technology, e.g. Corba, Web Services or .NET. Thus, interoperability between systems has become a serious problem.

In 2001, the OMG introduced a new approach to software development process called the Model-Driven Architecture (MDA)[1] Framework [ORM01]. The main goal of MDA is to solve the interoperability problem by lifting up the abstraction levels by models and generate code automatically from these models [KWB03, Fra03, FP04]. Above that, the framework should fill in the software development life cycle with an integrated set of standards in order to enable e.g. better tooling. The standards embraced by MDA include

[1] It is called model-driven because it provides a means for using models to understand, design, construct, deploy and maintain systems [OMG03].

MOF as the metamodeling language, UML as the modeling language, QVT as the transformation technique and XMI as the exchange format.

MDA allows a developer to focus on essential features of a system while modeling. In its essence, models are the only artefacts within the MDA software development process. Depending on its abstraction level, models are categorized in Platform-Independent Model (PIM)[2] and Platform-Specific Model (PSM).

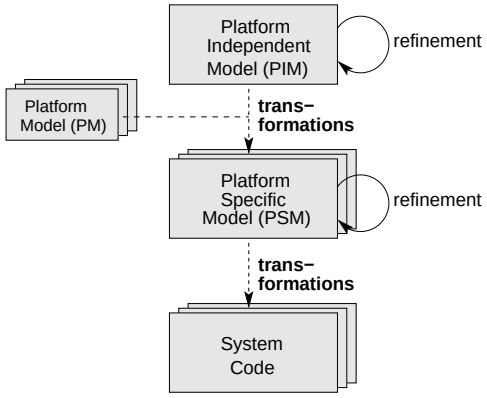


Figure 2.1: The MDA Framework

Following the MDA Framework, the initial model is always specified platform-independently. Model transformation provides a key technology within the MDA to bridge the semantic gap between models. A transformation defines a set of rules used to modify a source model to a target model. A minor transformation step is a refinement if it describes the relationship between models of the same abstraction level regarding the platform. By means of transformation and refinement techniques, platform-specific artefacts are added to the models and compiled to the final system code. Transformation and refinement steps are utilized as follows in the framework (Figure 2.1):

After having specified an initial PIM model, extentions and corrections on the model may be required. The PIM-to-PIM refinement step modifies models without requiring additional platform-specific information. To add platform-specific information to the PIM, the PIM-to-PSM transformation is performed. It describes how the functionality of the PIM are to be realized in terms of the PSM and its underlying implementation technology. For each platform where system code should run on, a PSM is generated from its PIM by means of a Platform Model (PM). A PM provides technical concepts and services provided by a certain platform. The PSM is

[2]In MDA, the term *platform* is used to refer to the technological details that are irrelevant to the functionality of the system component [OMG03].

improved by PSM-to-PSM refinement steps. Finally, the PSM-to-System-Code transformation step generates the system code, e.g. C++, Java etc, for the target platform. This last transformation step is comparable with the traditional compilation step. Of course, the completeness of the system code depends on the completeness of the models [DGL+03a, DGL+03b]. In Figure 2.1, one intitial PIM is shown. Additionally, PMs for three different platforms are provided. By means of model transformation, PSMs for each platform are derived. At last, system code are generated for the determined platforms.

In the following, we shall look into the languages and technologies provided and recommended by the MDA Framework in more detail.

2.1.1 The Meta-Object Facility (MOF)

The Meta-Object Facility (MOF) is a standardized technology initiated by the OMG in 1997. It is an extensible model-driven integration framework for defining, manipulating and integrating metadata and data. It provides a key technology for modeling languages within the MDA Framework. The current version, MOF 2.0 [OMG04] has been adopted by the OMG in 2004. Since then, a number of technologies standardized by OMG, including the UML 2.0, Common Warehouse Metamodel (CWM) and XMI use MOF 2.0 for metadata-driven interchange and metadata manipulation. According to MDA, models should be expressed in a MOF-based language. This guarantees that models stored in a MOF-compliant repository can be parsed and transformed by MOF-conformant tools [Ecl06, Con06, Omo06, Tes05, Med06, GR03, BSM+03].

The MOF 2.0 standard is divided into two subsets called the essential MOF (eMOF) and complete MOF (cMOF). eMOF is quite simply defined. It provides three packages to describe three mechanisms called *reflection*, *identifier* and *extension*: First, the reflection mechanism extends a model with the ability to be self-contained, i.e. it can be recursively defined by itself. Second, an identifier provides an extension for uniquely identifying metamodel objects without relying on model data. Third, the extension is a means for extending model elements with names and values. cMOF is highly coupled with the UML 2.0 Infrastructure (Section 2.1.2) and reuses its import and merge packages.

In order to understand the relationship between the various OMG standards within the MDA framework, we have to understand how metamodeling works. In its essence, metamodeling allows the definition of modeling languages in a unified way. Here, the OMG uses a so-called meta-layer architecture for its standards. The principle of the meta-layers is elementary: The lower layer is always an instance of the upper layer whilst the upper layer provides the metamodel for its lower layer. Figure 2.2 shows an ex-

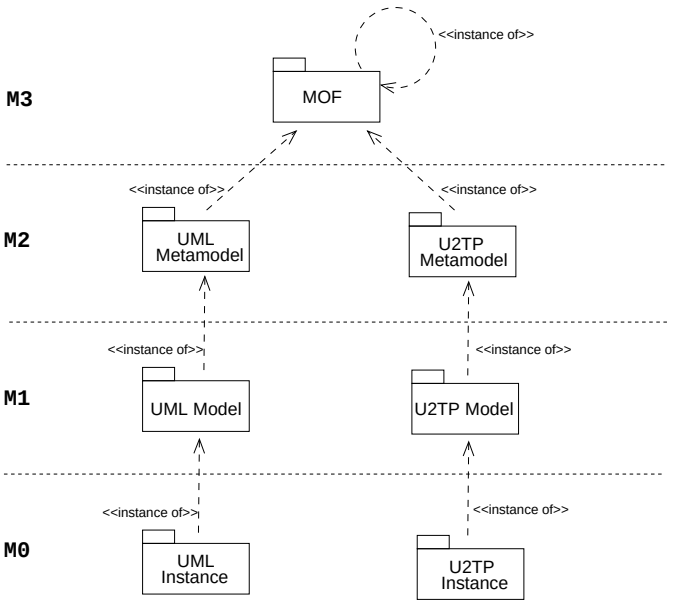


Figure 2.2: A Four-Layer Metamodel Hierarchy

ample with four meta-layers[3]. In the example, MOF is refered to as the meta-metamodel in Layer M3, meaning that all metamodels defined in its lower layer are instances of MOF elements. Above that, MOF may instantiate itself recursively based on its reflection mechanism. Two metamodels called UML and U2TP are shown in Layer M2. Along the same line, an element of the UML model in Layer M1 is an instance of an element of the UML metamodel in M2. Likewise, a concrete UML instance in the lowest layer M0 is an instance of its UML model in M1.

2.1.2 The Unified Modeling Language (UML)

Each MDA-based specification defines a platform-independent model and one or more platform-specific model(s). According to the MDA Framework, these models are defined in UML, making OMG's modeling language a foundation of the MDA. In 1997, the OMG adopted the first version of UML. Since then, UML has become popular for software modeling in both research and industrial worlds. UML helps its user to specify, visualize, and document models of software systems, including their structure and design to meet the system's requirements. A UML model captures information about the static structure and dynamic behavior of a system.

[3]While MOF 1.3 still specifies a four-layer metamodel hierarchy, MOF 2.0 does not determine the number of meta-layers any more. It only defines the relationship between meta-layer and its underlying layer [OMG04].

The current version, UML 2.0, has become an official OMG standard since October 2004. Tailored to the MDA requirements, UML 2.0 improves business, architectural, structural, and behavioral modeling. Inspired by modeling languages such as Specification and Description Language (SDL), Message Sequence Chart 2000 (MSC-2000) or Enterprise Distributed Object Computing (EDOC), UML 2.0 has been changed and improved [EHS97, (OM05, KSB03, CS03, GSDH97, GKSH99a, ANM97, EGH+97, HKN01, RR01, RGG99]. On its specification level, the UML 2.0 metamodel has completely been changed and restructured [FSG06]. On the user level, the language has become much more complex and powerful. Just to name some added features: composition of interactions in Interaction Diagrams and of states in State Machines is allowed; simple time concepts in Interaction Diagrams exist to retrieve time value and to constrain time between events; definition of internal class structure has been added; UML 2.0 profiling mechanism has been extended, etc [Mil03, Sel05].

The UML 2.0 standard is published in two complementary documents: the *UML 2.0 Infrastructure* and *UML 2.0 Superstructure* [UML04, UML05]. In the following, the main concepts defined in both documents are outlined.

UML 2.0 Infrastructure

The UML 2.0 Infrastructure defines the foundational language constructs required for UML 2.0. It consists of two packages called *Core* and *Profiles*, where the former contains main concepts used for metamodeling, and the latter defines mechanisms used to customize metamodels.

Basically, the Core package enables its reuse for new MDA metamodels to benefit from abstract syntax and static semantics of existing metamodels. The Core package is also considered as the architectural kernel of MDA since it provides fundamental concepts to define both the UML 2.0 and MOF metamodels.

A UML 2.0 profile is a domain-specific extension of UML 2.0 using standardized extensibility mechanisms[4]. A profile definition is based on referencing to certain existing metamodel elements. It is a special package which contains model elements that have been customized for a specific domain or purpose using the extensibility mechanisms. UML 2.0 defines three mechanisms to extend a metamodel [UML04]:

1. *Constraints*: Constraints are semantic conditions or restrictions. A constraint can be expressed in natural language text, in a mathematically formal notation, or in a machine-readable language for the purpose of declaring some of the semantics of a model element.

[4]By means of the extensibility mechanisms, Domain-Specific Languages (DSLs) can be defined by the user.

2. *Tagged Values*: Tagged values are explicit definitions of a property. In a tagged value, the name is referred as a tag. Normally, tags are defined by the user. Nevertheless, there are also predefined tags provided by UML 2.0.

3. *Stereotypes*: A stereotpye is a meta-class that defines how an existing meta-class may be extended. Stereotypes enable the use of platform or domain-specific terminology. Stereotypes are either predefined or user-defined.

In the following, the extension mechanism with stereotypes is examplified since it is intensively used in this work:

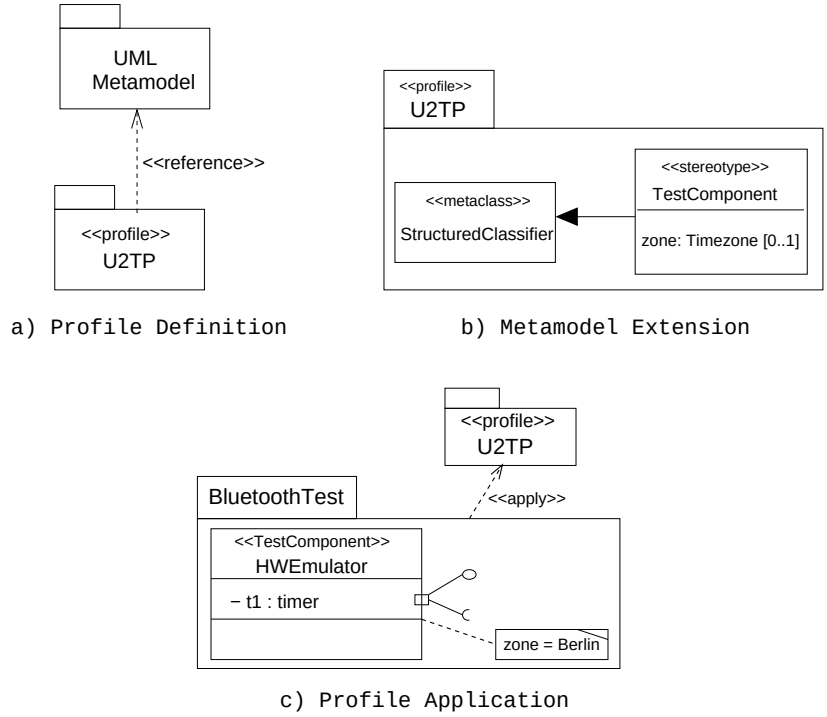


Figure 2.3: UML 2.0 Profiling Mechanism with Stereotypes

Figure 2.3 depicts how a UML 2.0 profile can be defined and applied to a model. In Figure 2.3a), the UML metamodel is defined in a package called UML. A new profile called U2TP references to the UML metamodel[5] in order to be able to extend the UML metamodel by new profile-specific concepts. The extension defined in the U2TP package is shown in Figure 2.3b). There,

[5]Referencing a metamodel is done by the dashed arrow attached by the stereotype <<reference>>.

a new stereotype of the U2TP profile called TestComponent extends the UML metaclass called StructuredClassifier by using the arrow with the solid arrowhead. Furthermore, the stereotype TestComponent has an attribute called zone of type TimeZone. When specifying a model based of a profile, all concepts of UML 2.0 can be used. Additionally, the newly defined profile concepts can be applied to the model. Figure 2.3c) shows the specification of a U2TP model, using the U2TP package concept to define a package called BluetoothTest. The reference to the profile package is indicated by the dashed arrow with <<apply>> as stereotype. A test component called HWEmulator is defined to emulate Bluetooth hardware where the *TestComponent* concept is a U2TP specific concept. Moreover, the test component owns a timer attribute called t1 and a timezone named Berlin attached to the test component class.

The UML 2.0 Superstructure

The UML 2.0 Superstructure defines user-level constructs and unifies various diagram types in order to be able to model all kinds of aspects of an object-oriented system [RJB05, JBR99]. Altogether, UML 2.0 defines thirteen types of diagrams, divided into three categories to represent static application structure, general behavior and different aspects of interactions: *Structure Diagrams* include the Package Diagram, Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, and Deployment Diagram. *Behavior Diagrams* include the Use Case Diagram, Activity Diagram, and State Machine Diagram. *Interaction Diagrams*[6] include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram. In the following, we will introduce part of the mentioned diagram types, which are most relevant for this work.

The *Package Diagrams* are used to divide a model into logical packages and to describe relation between the packages at a high level. Typically, Package Diagrams are used to organize the model. When used to represent class elements, Package Diagrams are utilized to provide a visualization of the namespaces. Elements contained in one Package share the same namespace. Packages may be imported from or merged with other packages. Figure 2.4a) shows a Package Diagram called thePackage comprising three packages called PkgA, PkgB and PkgC. PkgA is merged with PkgC, meaning that all elements defined in PkgC are included into PkgA. Furthermore, PkgB is imported by PkgA.

Class Diagrams depict the static view of the model or part of a model, describing the attributes and behavior the classes own. They are most useful to

[6]The UML 2.0 Superstructure standard [UML05] defines *Behavior Diagrams* and *Interaction Diagrams* explicitely in two categories even though the latter also model system behavior.

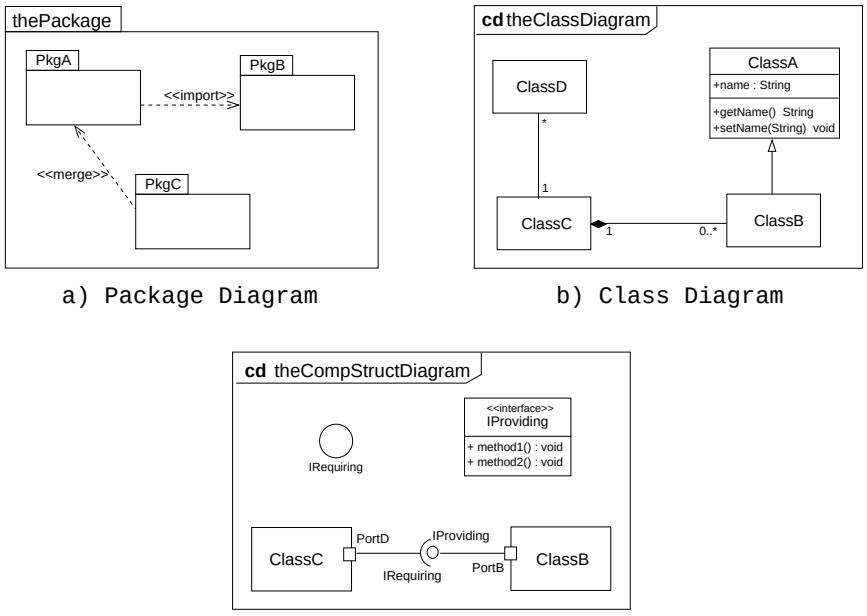


Figure 2.4: A Selection of UML 2.0 Structure Diagrams

illustrate relationships between classes and interfaces. A class is an element including its attribute and behavior definitions. An interface is a specification of behavior that implementers agree to meet. By realizing an interface, classes are guaranteed to support a required behavior. Additionally, generalizations, aggregations and associations are useful in order to reflect inheritance, composition and connections. A generalization is used to indicate inheritance and aggregations are used to depict elements which are made up of smaller components. An association implies two model elements having a relationship where its connectors may have roles, cardinalities, directions and constraints. Figure 2.4b) illustrates a Class Diagram called theClassDiagram, defining the four classes named ClassA, ClassB, ClassC and ClassD. ClassA comprises name as its attribute and a getName method returning a string and a setName method. ClassB inherits all attribtes and methods from ClassA. ClassC is a composition of ClassB elements. If the parent, in this case ClassC, of a composition is deleted, all of its parts are deleted with it. Furthermore, ClassC is associated with ClassD in a 1:n relationship.

Composite Structure Diagrams specify the internal structure of a classifier, including its interaction points to other parts of the system. It shows the configuration and relationship of parts which represent a set of one or more instances owned by a classifier instance. Ports define the interaction be-

tween a classifier and its environment. It provides or requires services on its interfaces. An interface is a class with restrictions. Interface operations are abstract. While a class inherits from a single super-class, an interface may implement multiple interfaces. Figure 2.4c) defines a Composite Structure Diagram with two classes ClassB and ClassC. The classes are connected to each other via connectors and the ports PortB and PortD. Additionally, interfaces are defined to provide or require services. ClassC specifies a requiring interface called IRequiring and ClassD defines a providing interface called IProviding. The definition of both interfaces are separately shown as a class element with <<interface>> as stereotype and in a circle, respectively.

State Machine Diagrams model the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime. A State Machine defines the various states a object might goes through. Figure 2.5a) specifies the behavior of an object called theStateMachine and the states the object goes through during its lifetime. This state machine owns three states called StateA, StateB and StateC. From the initial state, StateA is triggered by the create event. After that the object may go to StateB or StateC according to the triggering event. In case the event is trigger1, the object goes to StateB and invokes an effect1 as the result of the transition. In case the triggering event is trigger2, the object goes to StateC and invokes an effect2. From state StateB, the object can be triggered to StateC after receiving trigger3. The object goes to its final state if destroy is invoked.

Sequence Diagrams show the communication between objects. A lifeline represents an individual participant in a Sequence Diagram. Messages within a Sequence Diagram can be complete, lost or found, synchronous or asynchronous[7]. Furthermore, for complex behavior specifications, combined fragments are allowed within Sequence Diagrams which allow e.g. the definition of alternatives, optionals, loops, etc. In UML 2.0, some simple time concepts are added to the standard. These concepts provide means to represent time and durations, as well as actions to observe and constraint time.

Figure 2.5b) illustrates the communication between the three lifelines a, b and c of the classes ClassA, ClassB and ClassC, respectively. In the beginning, lifeline a sends message1 to lifeline b and waits for message2 as a response from lifeline b whereas the response should not last more than 3ms. After having answered to a, lifeline b sends message7 with a parameter to lifeline c and stores its return value. Afterwards, a message is lost by lifeline c and the lifeline is destroyed. Meanwhile, lifeline a has received message3 with a parameter from b. The time of sending message3 is recorded by lifeline b by the now operation and the message should be received by lifeline a within 3 seconds, indicated by "t..t+3". Afterwards, an alternative behavior is invoked between a and b: If the variable x=true, then message4 will be sent

[7] A synchronous message completes with an implicit return message whilst an asynchronous message waits for its explicit return message.

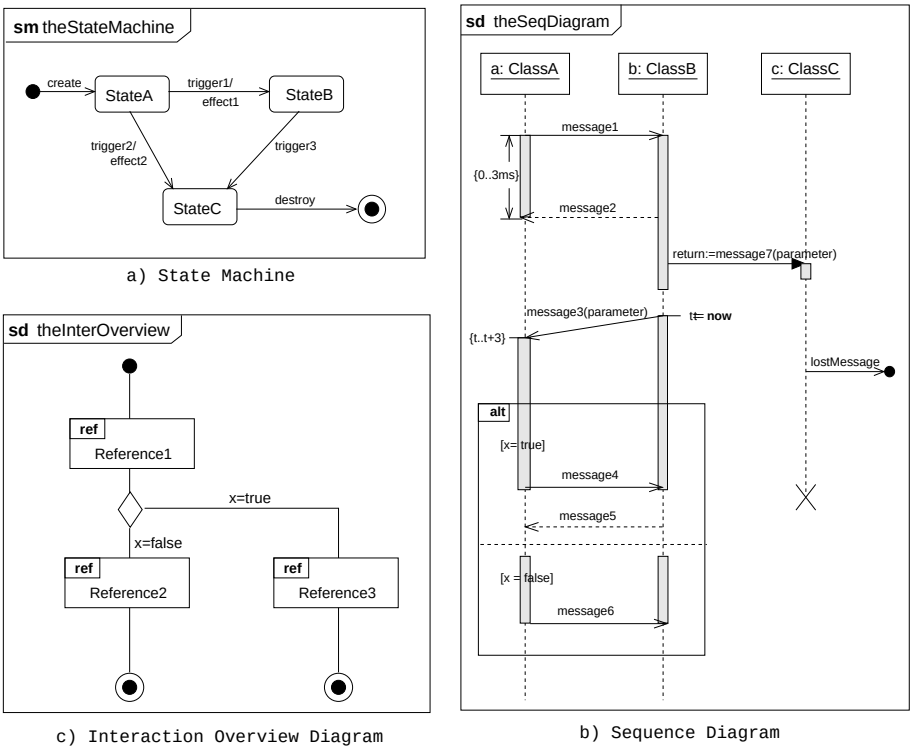


Figure 2.5: A Selection of UML 2.0 Behavior Diagrams

to lifeline b and message5 can be received. In case that the variable x=false, only message6 will be sent to lifeline b.

Interaction Overview Diagrams use Activity and Sequence diagrams to allow interaction fragments to be combined with flows and decision points. Interaction diagrams may include Sequence Diagrams, Communication Diagrams, Interaction Overview Diagrams or Timing Diagrams. However, this diagram type introduces two concepts called the Interaction Occurrence and the Interaction Element. While Interaction Occurrences are references to existing interaction diagrams, the Interaction Elements display the contents of the reference diagram inline. A simple Interaction Overview Diagram can be found in Figure 2.5c), referencing to three external diagrams called Reference1, Reference2 and Reference3. These references may be either a Sequence Diagram, a Communication Diagram, an Interaction Overview Diagram or a Timing Diagram. Furthermore, there is a decision made after the execution of Reference1. Accoring to the result of the variable x, Reference1 is followed by the execution of Reference2 or Reference3. After that, the behavior goes to its final node.

2.1.3 The Queries/Views/Transformations (QVT)

In 2002, the OMG requested proposals to define a language for specifying queries on and transformations between MOF 2.0 models. Generally, model transformation can be catagorized in three styles [DGL+00, CH03, VVP]:

1. *Source-Driven Transformation*, where the transformation rule is a simple pattern. The matched elements are transformed to a set of target elements. This style is often used in high-level to low-level transformations (e.g. compilation).

2. *Target-Driven Transformation*, in which a transformation rule is a complex pattern of source elements. The matched elements are transformed to a simple target pattern. This style is often used for reverse engineering or for performing optimizations.

3. *Aspect-Driven Transformation* describes the relationship of the semantic concepts, e.g. transforming all imperial measurements to one metric measurement, or replacing one naming system with another naming system etc.

The Request For Proposal (RFP) published by OMG is called the Queries/-Views/Transformations (QVT) [OMG02a]. QVT is requested to offer a technology for model-driven development of large distributed systems. In its essence, the submission requires the definition of a transformation language which describes relationships between a source and a target MOF 2.0 metamodel. Also, it should allow the creation of views and querying of metamodels. The submission should ensure that incremental changes to source models can be propagated and new languages can be expressed as MOF 2.0 models. Several proposals have been submitted to the request since then.

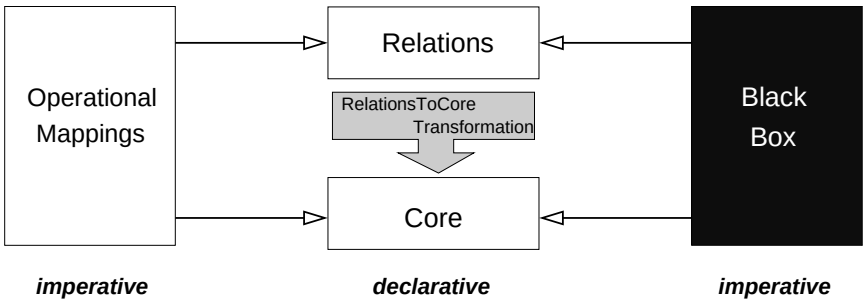


Figure 2.6: QVT Language Parts

At time of writing this thesis, the standardization process of QVT is still ongoing. The current status of the QVT submission can be summed up as follows [OMG05]: it has a hybrid declarative/imperative nature, with the declarative part being split into a two level-architecture (Figure 2.6):

The declarative language part is structured into a two-layer architecture called *Relations* and *Core*. They are mainly responsible for pattern matching and tracing. The *Relations* language is a declarative specification of the relationships between MOF models. It supports complex object pattern matching, and implicitly creates trace classes and their instances to record what occurred during a transformation execution. Relations can assert that other relations also hold between particular model elements matched by their patterns. The *Core* language is a small language supporting pattern matching by evaluating conditions over variables against a set of models. It treats all of the model elements of source, target and trace models symmetrically. The trace models in *Core* must be explicitly defined, and are not deduced from the transformation description.

In addition to the declarative languages, there are two imperative language parts specifying normal programming languages with mechanisms to invoke the imperative transformation implementations from the declarative parts: the standard language called *Operational Mappings* and the non-standard *Black-Box* implementation. The *Operational Mappings* language is specified as a standard way of providing imperative implementations, which has the same trace models as the *Relations* language. Mappings Operations can be used to implement one or more Relations when it is difficult to provide a purely declarative specification. *Black Box* implementations may be derived from Relations making it possible to "plug-in" any implementation of a MOF Operation with the same signature. This allows complex algorithms to be coded in any programming language with a MOF binding as well as the use of domain specific libraries to calculate model property values.

Tefkat - A Transformation Language

Various approaches on model transformation exist. Transformation languages with appropriate underlying engines like MTF [IBM06a], Atlas Transformation Language (ATL) [ATL06], Action Semantics Language (ASL) [Car03], Medini [Med06] or Tefkat [Tef06] etc. give several examples. In this section, the transformation language called *Tefkat* and its transformation engine is introduced. Actually, there is no tool yet available compliant to the current QVT standard. Tefkat is one of the pioneering transformation languages to QVT. The experience gained from Tefkat has been integrated into the QVT standardization. We chose Tefkat as our transformation language because it is intuitive and there is an Eclipse plugin implemented for the Tefkat engine which is user-friendly and well-supported [Tar06][8].

[8] At the time of this work, Tefkat was our first choise. Of course, the development of

Tefkat is an aspect-driven, declarative transformation language [Tef06]. Together with a transformation engine, it has been developed by the Distributed System Technology Centre (DSTC). We shall use Tefkat for the model transformation approach in a later chapter of this work.

In its essence, the Tefkat transformation language defines three main concepts [Tar06, DGL+00]: *transformation rules*, *patterns* and *tracking relationships*. Transformation rules are used to describe correspondence between elements in a source model and the elements to be created in the target model. Patterns are expressed as reusable definitions. When used in the source of a rule, a pattern is a query. When used in the target of a rule, it acts as a template for the model elements. Tracking relationships are used to associate the source model elements with the target model elements whose existence implies a rule [DGL+03a, DGL+03b].

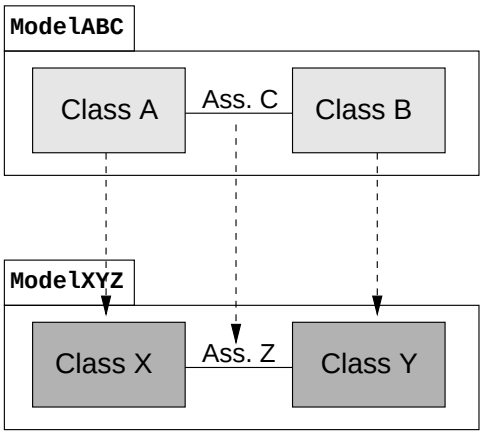


Figure 2.7: Transformation Principles

Figure 2.7 depicts the principles of the Tefkat model transformation. In this simple example, two packages are shown. Each package owns two classes and one association. In order to transform from one package to the other, three transformation rules have to be defined: one to transform Class A to Class X, another for Class B to Class Y, and a third transformation rule for the association. The first two rules are easily written, but the third must be able to refer to the instances of Class X and Class Y that were created from Class A and Class B so that the newly created link of type Association Z has the correct objects at each end. This is done by using tracking relationships in the rules for transforming the classes and referring to the relationships in the rule for the association. In the following, language constructs of Tefkat are explained in detail[9]. They are needed in order to understand the transformation rules defined in this work.

the other languages and engines mentioned above has also improved since then.

[9]The language is explained by means of its BNF

```
1 TRANSFORMATION name : source [, source]* -> target [, target]*
2
3 IMPORT uri
```

Figure 2.8: Tefkat Language Construct: Transformation and Import

A Tefkat *transformation* specification is normally stored in a single file with the suffix **.qvt*. This file starts with a line of the form shown in line 1 of Figure 2.8. It provides a name for the transformation followed by names for each of the source model instances and the target model instances. In order for the transformation engine to be able to resolve model class names, the location of models must be specified. This is done by the **IMPORT** statement in Line 3. Generally, the URI is of the form "http://...", "platform:/resource/..." or "file:/...".

```
1  RULE name
2          FORALL class variable
3          WHERE condition
4          MAKE (class variable FROM name (expressions)
5                | template_name (expressions)
6                )
7          SET path = expression
8          LINKING tracking
9          WITH name = expression
10 ;

11 RULE name (vars)
12         EXTENDS name (vars) [ , name (vars) ]*
13       | OVERRIDES name (vars) [ , name (vars) ]*
```

Figure 2.9: Tefkat Language Construct: Rule Definition and Rule Extention

In Tefkat, a **RULE** consists of the following optional sub-parts: **FORALL**, **WHERE**, **MAKE**, **SET**, and **LINKING** (Figure 2.9, Lines 1–10). The clauses **FORALL** and **WHERE** are used on the source model elements to query all elements which have the same features defined in the **WHERE** clause. **MAKE** and **SET** are used on target model elements to create a new object and to set the attribute values of the newly created object. **LINKING** is required in order to allocate tracking relationships of the model elements. A rule can extend or override other rules, but may not both extend and override the same rule (Figure 2.9, Lines 11–13).

```
1 PATTERN name (variables)
2         FORALL class variable
3         WHERE condition

4 TEMPLATE name (variables)
5         MAKE class variable
6         WHERE condition
7         LINKING tracking WITH name = expression
```

Figure 2.10: Tefkat Language Construct: Patterns and Templates

PATTERNs and **TEMPLATE**s are rules with limitations. While patterns only match to source model elements, templates are limited to match to target model elements. Both patterns and templates may be parameterized (Figure 2.10).

CONDITIONS take on the usual form of boolean-valued comparisons or pattern invocations linked together by operators **AND**, **OR** and optionally prefixed by **NOT** (Figure 2.11, Lines 1–4). **UNDEF** is used to test for a null-valued feature. **EXPRESSION**s are either variable names, literals, function calls, path expressions, or expressions involving usual mathematical operators (Figure 2.11, Lines 5–10).

```
1  ( expression op expression
2      | name (expressions)
3      )
4      [(AND | OR) condition ]*

5   "string"
6      | integer
7      | true | false
8      | <uri>
9      | name ( expressions )
10     | variable  [( . | -> )  feature ()? {}?]+
```

Figure 2.11: Tefkat Language Construct: Conditions and Expressions

The underlying transformation engine of Tefkat is available as an plugin for the Eclipse platform[10] [Ecl06]. The metamodels on which transformation should be performed must be defined in a MOF-like meta-metamodel

[10]Eclipse is an open development platform and application frameworks for building software.

structure of Eclipse called Eclipse Modelling Framework (EMF) and transformations are then performed between these EMF meta-classes [GR03].

2.2 Test Concerns

Testing is one of the most important phases during the software development process with regard to quality assurance. It is an analytic means for assessing the quality of software [Wal01]. While Myers defines testing as the process of executing a program with the intent of finding errors [Mye79], Balzert regards testing as a means aiming at revealing errors in a program [Bal98]. In this section, important testing aspects and test development phases are explained. Moreover, an insight into the standardized test language called TTCN-3 is given.

2.2.1 Test Dimensions and Test Development Phases

Software or communication systems can be classified in different categories according to its communication to its environment, the distribution of system components, and the real-time aspects [Gra02]. Similarly, tests can also be classified in different levels, depending on the characteristics of the system under test and the test system. [Neu04] categorizes testing in the dimensions of test goals, test scope and test distribution, as shown in Figure 2.12.

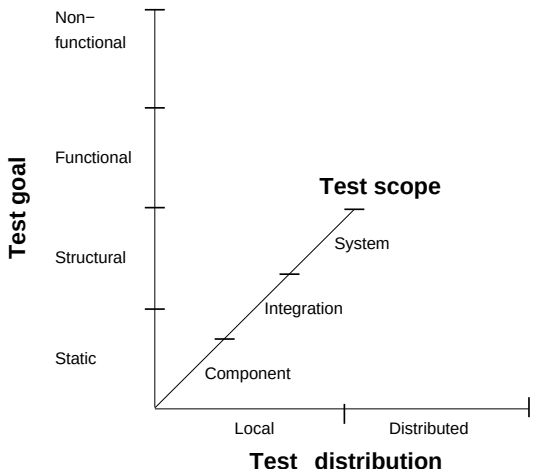


Figure 2.12: The Three Test Dimensions

Test goals The goal of a test can be categorized into *static* and *dynamic* testing, whereas the latter is distinguished between structural, functional and non-functional testing.

1. *Static testing:* Static tests assess a system under test without executing it. They can locate defects in an early stage.

2. *Structural testing:* Structural tests cover the structure of the system under test during test execution (e.g., control or data flow). To achieve this, the internal structure of the system needs to be known. Therefore, structural tests are also called *glass-box tests* [Mye79].

3. *Functional testing:* The goal of functional testing is to ensure the correctness of the system under test with respect to its functionality according to the system specification. In contrast to structural tests, functional tests do not require any knowledge about system's internals. They are therefore called *grey-box tests* or *black-box tests* [Bei95].

4. *Non-functional testing:* Similar to functional tests, non-functional tests are performed against requirements specification of the system. In contrast to pure functional testing, non-functional testing aims at the assessment of non-functional, such as real-time requirements. Non-functional tests are usually black-box tests. Nevertheless, for retrieving certain information, e.g. internal clock, internal access is required. In this case, these tests are also regarded as *grey-box tests*.

Test scope *Test scopes* describe the granularity of the system under test. Due to composition of the system, testing at different scopes may reveal different failures [Wey86, Wey88]. Therefore, tests on different scopes are usually performed in the following order:

1. *Component:* At the scope of component testing, the smallest testable component (e.g., a class in an object-oriented implementation or a procedure in a procedural language) is tested in isolation.

2. *Integration:* The scope of integration test is to combine components with each other and test those not yet as a whole system but as a subsystem.

3. *System:* In a system test, the complete system consisting of subsystems is tested. A complex system may be distributed and has usually different interfaces through which it can be accessed and tested.

Test distribution Not only the System Under Test (SUT) may be distributed, but also the test system itself can be characterised with respect to its distribution:

1. *Local:* A local test consists of just one test component located on a single node. The test is driven by the test component which accesses the SUT through one or more interfaces.
2. *Distributed:* A distributed test consists of several test components which may be distributed over multiple nodes. Thus, the whole test consists of concurrently running components which interface the item under test. To achieve a deterministic test, coordination and synchronization between the components is necessary.

For this thesis, we modified the dimensions shown in Figure 2.12. Since the work deals with both local and distributed systems, the test distribution dimension becomes trivial. Thus, we replaced this dimension by a dimension describing the different test development phases (Figure 2.13). The aim of this work is to cover as many categories over the test development phases as possible.

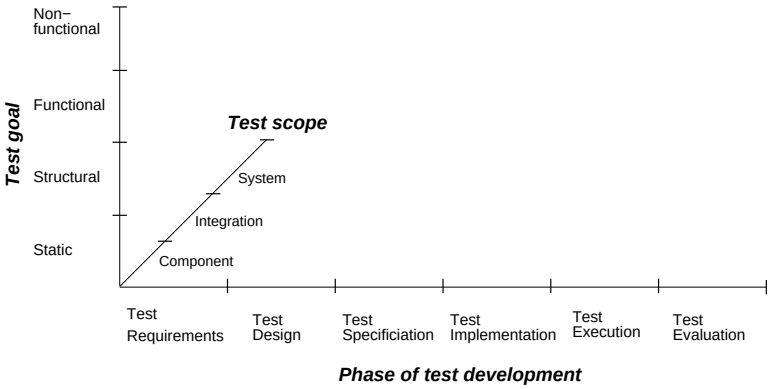


Figure 2.13: Application Areas and Test Dimensions

Phase of Test Development The dimension of the *phase of test development* relates to the point in time of the test development process. The test development process can be divided into the following phases which are usually consecutive, but may be iterated as part of an incremental or iterative development approach:

1. *Test requirements:* Preceeding to the actual test development, test requirements need to be captured.

2. *Test design:* The design of a test mainly concentrates on the test architecture and its involved classes. Depending on the test distribution, i.e. local or distributed, the number of test components and their connections are defined.

3. *Test specification:* During test specification, both test architecture and test behavior are concretized. For test behavior specification, communication between the SUT and test components is defined and test coordination between components of test system must be determined. Also, concrete test data are needed. Furthermore, timers provide good means to control and guarantee the proper termination a test.

4. *Test implementation:* This phase realizes the tests specified in the previous stage to make it executable. Herefore, implementation specific information, such as the system infrastructure, port mapping between SUT and the test system, or data coding must be taken into account.

5. *Test execution:* In the test execution step, the tests are executed automatically or manually. During test execution, test events may be logged for further evaluation.

6. *Test evaluation:* The final phase of the testing process is the evaluation of test results which is performed after test execution, eventually.

2.2.2 Real-Time Properties and Testing

Black-box testing is performed against requirements that can be classified in functional and non-functional testing. In the following, the difference between both is addressed and real-time properties are discussed. Partsch defines a requirement firstly, as "a condition or capability needed by a user to solve a problem or achieve an objective", and secondly "as a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. The set of all requirements forms the basis for subsequent development of the system or system component" [Par91].

Functional requirements are associated with tasks or behavior a system must support, while non-functional requirements are constraints on various attributes of the functional behavior[11]. Thus, non-functional properties are always related to functional behavior and do not exist on their own. A brief classification of requirements is outlined in Figure 2.14. As depicted there, non-functional requirements can be subdivided with respect to the following properties: *real-time* properties; *reliability* properties like time Mean Time

[11] In the context of services, most of the non-functional properties are also denoted as Quality of Service (QoS).

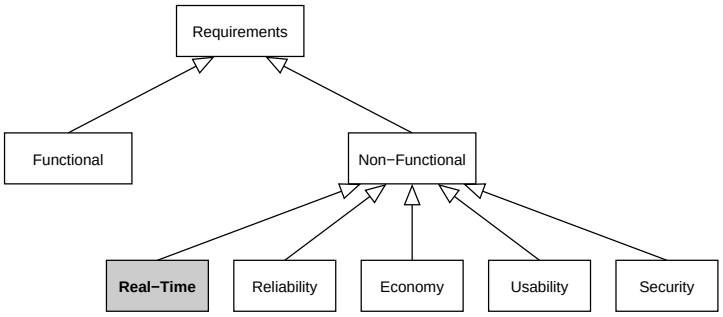


Figure 2.14: Hierarchy of Requirements

Between Failure or robustness; *economic* properties, e.g. costs or maintainability; *usability* properties and *security* properties. In this thesis, we focus on real-time properties.

Real-Time Systems and Real-Time Properties According to Lann, a real-time system is "a computing system where initiation and termination of activities must meet specified timing constraints" [lL90]. Concerning the timing constraints a real-time system has to comply, *hard* and *soft* real-time properties can be distinguished [SR90]:

Hard real-time properties are time constraints which have to be fulfilled in any case. If the hard real-time constraint is, e.g. defined on the duration of a request response, a late answer indicates a failure in the system. An example gives the autopilot of an aircraft, where violation of hard real-time constraints might lead to a crash. Mathematically, hard real-time properties can be described by simple equations or inequations.

Soft real-time properties are time constraints which need to be satisfied only in the average case or to a certain percentage. In this case, a late answer is still a valid answer. An example is video transmission where a delayed frame might either be displayed delayed or dropped when not perceivable as long as no consecutive frames are affected. Even though soft real-time systems are easier to implement, the mathematical description of soft real-time properties is more complex, since it often involves statistical expressions.

In this thesis, we will address real-time properties such as delays *latency* and *response time* but also *frequency* or its reciprocal cycle duration *periodicity* as well as *throughput* [ATM99a, ATM99b, ATM00, RB96, IET90, IET91, IET98, IET99, IET02].

Functional vs. Real-Time Testing Functional testing is concerned with assessing the functional behavior of an SUT neglecting time aspects[12]. While the temporal order of events is regarded, the exact timing of the events is not considered. Functional testing can be used for assessing both real-time and non-real-time systems.

In contrast, real-time testing assesses the real-time properties of an SUT by considering the points in time when test events are exchanged. This fact does not only apply for observations. Also stimuli which are sent by the test system may be subject of real-time requirements, since the SUT might also assume certain real-time properties from the environment, provided by the test system while testing.

Indeed, since non-functional properties like real-time properties are always related to functional behavior, they cannot be tested on their own, but require a test case which involves also the associated functional events for stimulating and observing the SUT. Thus, a prerequisite of real-time testing is that the SUT has successfully passed the functional testing.

An important requirement on testing is that tests shall be deterministic and reproducible. Otherwise, if the test verdict changes for each execution of the same test case, the outcome of a test run would be of no relevance. However, reproducibility is a severe problem for real-time testing [Sch93]. Problems occur if the SUT delivers a real-time performance which is close to the limit of the actual real-time requirement, i.e. if the real-time property varies below and beyond a certain limit. In this case, it is difficult to decide whether the SUT really failed or whether the test system is in fact at its limits and thus responsible for non-deterministic observations[13]. This can be avoided if the test system itself obeys two requirements: Firstly, the time resolution of the clock used to assess the test item's real-time property is at least in the same order of magnitude as the real-time requirements which are subject of test. Secondly, the test system in general has to be fast enough to send the stimuli and make the observations in time. Hence, to be able to execute real-time tests, an important prerequisite is that the test system is fast enough. This can, e.g. be evaluated by benchmarking the test system itself [HKN01, DST04] or using real-time operating systems for the test infrastructure [Mag00]. Generally, for real-time testing one has to be aware that not only the SUT is a real-time system, but also does the test system. Hence, for the implementation of a real-time test, real-time support from a real-time operating system such as *RTLinux* or *FreeRTOS* is required [RTL06, Fre06]. However, test implementation on such a low level is out of the scope of this thesis.

[12]Though time aspects can be neglected in functional teesting, the use of timers is still required.

[13]In worst case, i.e., the test system is far too slow, a test may be deterministic in the sense that it yields always the wrong verdict.

The Tree and Tabular Combined Notation (TTCN) is a language for conformance test specification. It is defined in a framework for functional black-box testing process, called the Conformance Testing Methodology and Framework (CTMF) [ISO97]. This language is intended for pure functional black-box tests. A more powerful successor of that test language, the TTCN-3 is described in Section 2.2.3.

2.2.3 TTCN-3 - A Testing Language

The Testing and Test Control Notation, version 3 (TTCN-3) is a test specification and implementation language to define test procedures for black-box testing. TTCN-3 [ETS05a, GWWH00] has been developed from 1998 to 2001 by an European Telecommunications Standards Institute (ETSI) experts team as successor language for TTCN. Its development was driven by the industry and research in order to obtain a single test notation for black-box testing needs. In comparison to TTCN, TTCN-3 provides several additional concepts like, e.g. dynamic test configurations, procedure-based communication and module control part.

By means of TTCN-3, a tester is able to specify tests at an abstract level and focus on the definition of the test cases rather than test system adaptation and execution. TTCN-3 enables a systematic and specification-based test development for various kinds of tests, including the functional, scalability, load, interoperability, robustness, regression, system and integration testing [Din04]. In the following, the core language of TTCN-3 is explained.

An Overview on TTCN-3

An overall view on TTCN-3 can be found in Figure 2.15: Despite of the textual TTCN-3 Core Language (CL), presentation formats can also be taken as front-ends of the language. The tabular and graphical formats called Tabular Presentation Format for TTCN-3 (TFT) and Graphical Presentation Format for TTCN-3 (GFT) are standardized formats of TTCN-3. Other presentation formats can be added according to the needs of the users. Furthermore, the core language of TTCN-3 provides interfaces to referenced data which are defined in other description languages. For that, types and values such as defined in the Abstract Syntax Notation One (ASN.1) or Interface Definition Language (IDL) can be imported to TTCN-3.

The current ETSI standard for TTCN-3 comprises the following seven parts:

1. The Core Language (CL) introduces test-specific language constructs and defines the syntax of TTCN-3 [WDT+05].
2. The Tabular Presentation Format for TTCN-3 (TFT) is one of the standardized presentation formats of TTCN-3 [ETS05b]. In TFT, a

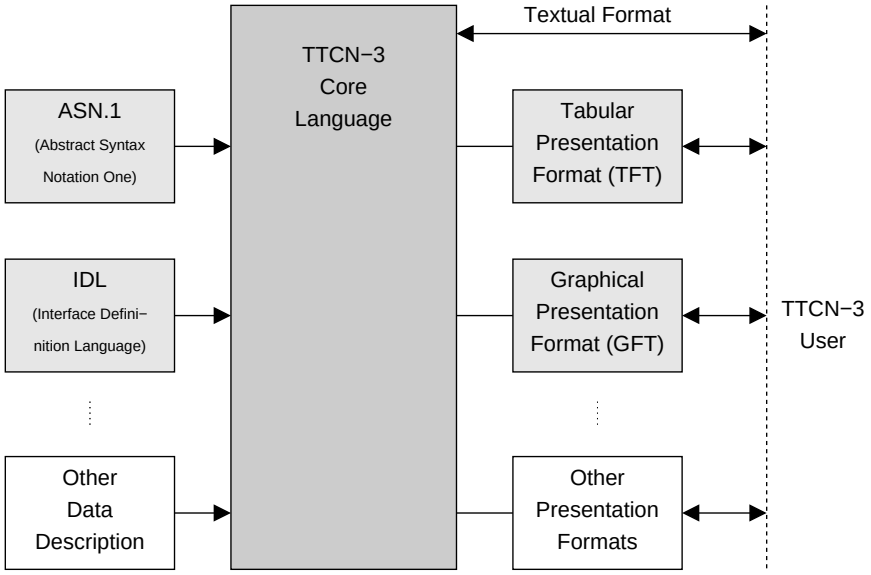


Figure 2.15: TTCN-3 Overview

TTCN-3 module is a collection of tables. TFT is similar in appearance and functionality to the earlier versions of TTCN [ETS98]. It is designed for users who prefer the old style of writing TTCN test suites.

3. The Graphical Presentation Format for TTCN-3 (GFT) is the second standardized presentation format [ETS05c]. This format is based on the style of Message Sequence Chart (MSC) [Z1299]. The graphical format is used to visualize TTCN-3 behavior definitions. The complete CL is mapped to GFT symbols and vice versa.

4. The Operational Semantics (OS) describes the semantic of TTCN-3 constructs and provides a state-oriented view of the execution of a TTCN-3 module [ETS05d].

5. The TTCN-3 Runtime Interface (TRI) provides a platform-specific adaptation layer to complete a test system implementation [ETS05e]. The TRI document contains the specification of a common Application Programm Interface (API) interface to adapt TTCN-3 test systems to the SUT.

6. The TTCN-3 Control Interface (TCI) also defines an adaption layer [ETS05f]. Furthermore, it is an implementation guideline for the ex-

ecution environments of TTCN-3. It contains the specification of the API which TTCN-3 execution environments should implement in order to ensure the communication, management, component handling, external data control and logging during test execution.

7. The last and newest document in the TTCN-3 standard defines the harmonization and use of ASN.1 with TTCN-3 [ETS03].

Concepts of TTCN-3

The TTCN-3 Core Language (CL) is a modular language and has a similar look of a common programming language. In contrast to common programming languages, TTCN-3 also contains concepts necessary to specify test artefacts like verdicts, timers, data matching, dynamic test configuration, encoding, synchronous and asynchronous communication etc. A TTCN-3 test specification typically consists of the following parts:

1. test data and templates definition,
2. function and test case definitions for test behavior, and
3. test control definition for the execution of test cases.

Modules The top-level building-block of a TTCN-3 test suite is a module. A module is defined with the keyword **module** and may import completely or partially the definitions of other modules. A module can be parameterized. Typically, a TTCN-3 module owns two parts: the module definition part and the module control part. The module definition part contains data and behavior defined by the module. They can be defined within the module, but also be imported from other modules. Test behavior in TTCN-3 are defined as **functions, altsteps** and **testcases**. The module control part specifies the main program and describes the test execution sequence. The control part calls test cases or functions defined in the module. Also, the module control part has access to test verdicts delivered by each test cases and decides about next test execution steps.

Figure 2.16 shows an example of a TTCN-3 module called MyModule. In its module definition part, this module imports all definitions from another module called OtherModule. Additionally, it specifies a type myType and a test case MyTestCase, where the latter is executed in the module control part.

Test System A test case is executed by a test system. TTCN-3 allows the specification of dynamic and concurrent test systems. A test system consists of a set of interconnected test components with well-defined communication

```
1  module MyModule {
2
3      /** Module definition part **/
4      import from OtherModule all; // imports from other modules
5
6      type record myType { // type definitions
7          charstring field1,
8          integer field2
9      }
10     /** Testcase definition **/
11     testcase MyTestCase(integer i)
12         runs on MyTestComponent system SysComponent {
13         // behavior definition
14     }

15     /** Module control part **/
16     control {
17         execute(MyTestCase(5))
18     }
19 }
```

Figure 2.16: Example on TTCN-3 Module

ports and an explicit test system interface which defines the boundaries of the test system.

Within every test system, there is one Main Test Component (MTC). All other test components are called Parallel Test Component (PTC)s (Figure 2.17). The MTC is created and started automatically at the beginning of each test case execution. A test case terminates when the MTC terminates. The behavior of the MTC is specified in the body of the test case defini-

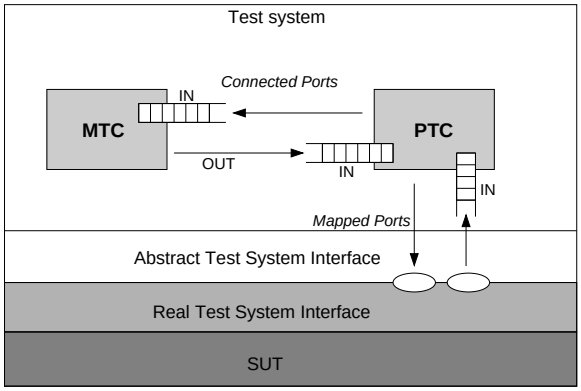


Figure 2.17: Conceptual View of a TTCN-3 Test System

tion. During the execution of a test case, PTCs can be created, started and stopped dynamically. A test component may stop itself or can be stopped by another test component. For communication purposes, each test component owns a set of local ports. Each port has an in- and out-direction. The in-direction is modeled as an infinite First In First Out (FIFO) queue, which stores the incoming information until it is processed by the test component owning the port. The out-direction is linked directly to the communication partner.

Figure 2.18 shows the definition of two port types. The first port type is a message-based port called MessagePortType and the second port type defines a procedure-based port called ProcedurePortType. The keywords **message** and **procedure** indicate the type of port.

```
1  type port MessagePortType message {
2      in MessageType1;
3      out MessageType2;
4      inout MessageType3
5  }
6
7  type port ProcedurePortType procedure {
8    out Proc1;
9    in Proc2
10 }
```

Figure 2.18: Example on Port Definition

During the test execution, TTCN-3 distinguishes between connected and mapped ports. Connected ports are used for the communication with other test components. If two ports are connected, the in-direction of one port is linked to the out-direction of the other, and vice versa. A mapped port is used for the communication with the SUT. TTCN-3 distinguishes between abstract and real test system interfaces. The abstract test system interface is modeled as a collection of ports that defines the abstract interface to the SUT. The real test system interface is the application-specific part of a TTCN-3-based test environment and implements the real interface of the SUT. In TTCN-3, connections between test components and mappings between test components and the SUT are created and destroyed dynamically at runtime.

An example of a system configuration definition can be found in Figure 2.19. Here, the test component MyComponent is created first. Afterwards, ports between the newly created test component and the MTC are connected. Also, a port of the test component is mapped to a port of the SUT. At last, the behavior of the test component is started.

```
1 var MyComponent := MyComponentType.create;
2
3 connect(self:Port1, MyComponent:port1);
4 map(MyComponent:Port2, system:SUTPort);
5
6 MyComponent.start(MyBehavior(self));
```

Figure 2.19: Test System Configuration

Test Cases and Test Verdicts Test cases define test behavior to check whether the SUT passes the test or not. The result of a test case execution is always a test verdict. TTCN-3 provides special test verdict mechanisms to interprete test runs. This mechanism is implemented by a set of predefined verdicts, local and global test verdicts and operations for reading and setting local test verdicts. The predefined verdicts are **pass, inconc, fail, error** and **none**. They are used for the judgment of complete and partial test runs. A **pass** verdict denotes that the SUT behaves according to its system requirement, a **fail** indicates that the SUT violates its specification, an **inconc** (inconclusive) describes a situation where neither a **pass** nor a **fail** can be assigned, and the **error** verdict indicates an error in the test system. The verdict **none** is the initial value where no verdict has been assigned yet.

Current value of verdict	New verdict assignment value			
	pass	inconc	fail	none
pass	pass	inconc	fail	pass
inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail
none	pass	inconc	fail	none

Figure 2.20: TTCN-3 Overwriting Rules for Verdicts

During test execution, each test component maintains its own local test verdict. A test component can retrieve and set its local verdict. When changing the value of a local test verdict, special overwriting rules are applied (Figure 2.20). The overwriting rules only allow that a test verdict becomes worse, e.g., a **pass** may change to **inconc** or **fail**, but a **fail** cannot change to a **pass** or **inconc**. In addition to local test verdicts, the TTCN-3 runtime environment maintains a global test verdict. The global test verdict is not accessible for the test components. It is updated according to the overwriting rules when a test component terminates. The final global test verdict is returned to the module control part when the test case terminates (Figure 2.21).

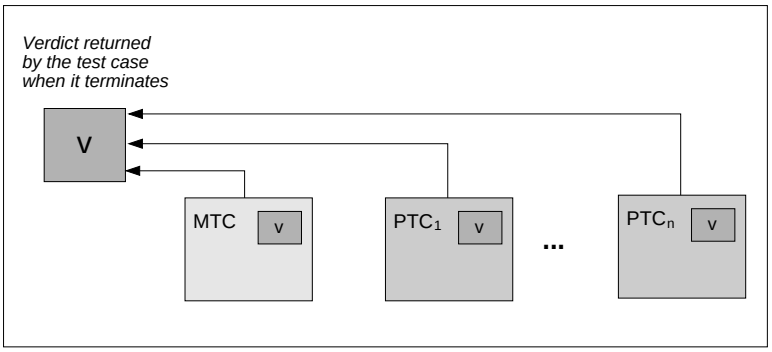


Figure 2.21: Relationship between TTCN-3 Verdicts

Alternatives and Snapshots A special feature of the TTCN-3 semantics is snapshots. Snapshots are related to the behavior of components. They are needed for the branching of behavior due to the occurrence of timeouts, the termination of test components and the reception of messages, procedure calls, procedure replies or exceptions. In TTCN-3, this branching is defined by means of alternatives, the **alt** statement.

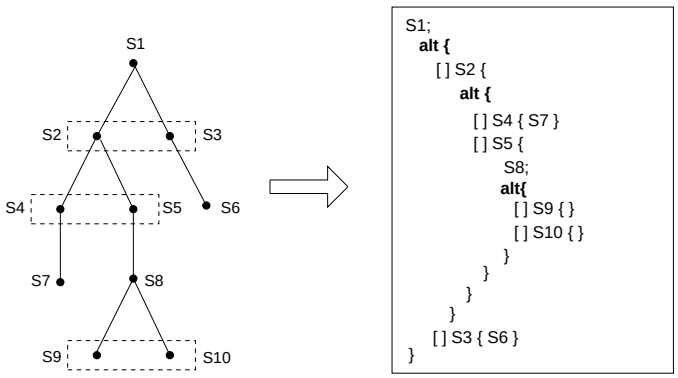


Figure 2.22: Illustration of TTCN-3 Alternative Behavior

An **alt** statement describes an ordered set of alternatives, i.e., an ordered set of alternative branches of behavior (Figure 2.22). Each alternative has a guard which consists of several preconditions referring to values of variables, status of timers, contents of port queues and identifiers of components, ports and timers. An alternative becomes executable if the corresponding guard is fulfilled. If several alternatives are executable, the first executable alternative in the list of alternatives will be executed. If no alternative becomes executable, the **alt** statement will be executed again.

The evaluation of several guards needs time. During that time, preconditions may change dynamically. This will lead to inconsistent guard evaluations, if a precondition is verified several times in different guards. TTCN-3 avoids this problem by using snapshots. Snapshots are partial module states which include all information necessary for the evaluation of **alt** statements. A snapshot is recorded when entering an alternative. For the verification of preconditions, only information in the current snapshot is used. Thus, dynamic changes of preconditions do not influence the evaluation of guards.

Default Handling Defaults are used to handle communication events which may occur, but which do not contribute to the test objective. Default behavior can be specified by **altsteps** and then activated as defaults. For each test component, defaults are stored in a default list according to the order of their activation. **Altsteps** are function-like descriptions for structuring component behavior. TTCN-3 uses **altsteps** to specify default behavior or to structure the alternatives of an **alt** statement. Like an **alt** statement, an **altstep** defines an ordered set of alternatives. The difference is that no snapshot is taken when entering an **altstep**. The evaluation of the top alternatives is based on an existing snapshot.

```
 1 altstep MyAltStep1() runs on MyComponentType {
 2 // altstep behavior
 3    [] MyPort1.receive(MyTemplate1) {
 4       setverdict(pass);
 5    }
 6    [] MyPort2.receive(*) {
 7       setverdict(fail);
 8    }
 9    [] MyPort2.catch(timeout) {
10       setverdict(inconc);
11    }
12 }
13
14 testcase MyTestCase() runs on MyComponentType {
15 ...
16    var default myDefault = activate(MyAltStep1());
17          ...
18         // test behavior with alternatives
19    deactivate(MyAltStep1);
20    }
21 }
```

Figure 2.23: Example on Default and Altstep Handling

The default mechanism is invoked at the end of each **alt** statement. The default mechanism invokes the first **altstep** in the list of defaults and waits for the result of its termination. The termination can be successful or unsuccess-

ful. Unsuccessful termination means that none of the top alternatives of the **altstep** defining the default behavior is executable. Successful termination means that one of the alternatives has been executed.

Figure 2.23 examplifies the definition of an **altstep** named MyAltStep1 and the activation of this default. The **altstep** defines three alternatives, catching the events of receiving MyTemplate1 via Port1, any events (represented by *) and timeout event via Port2 and a timeout event. According to the received event, verdicts are set.

Communication Operations Communication operations are important for the specification of test behavior. TTCN-3 supports message-based and procedure-based communication. The communication operations can be grouped in two parts: stimuli, which send information to SUT, and responses used to describe the reaction of the SUT.

Communication between test components and SUT can be asynchronous or synchronous. The communication is called asynchronous when messages are sent to or received from the SUT. It is called synchronous when procedures of the SUT are called by test components. Also, operations may be non-blocking, meaning that the test system does not wait for response after the call. Otherwise, the system is blocked until the response arrives. Figure 2.24 shows an example of the usage of communication operations.

```
1 MyPort1.send(myValue);
2 MyPort2.send(myValue) to MyComponent1;
3 MyPort3.call(MyProc:{MyVar1}) to MyComponent2 {
4     [] MyPort3.getreply(MyProc:{MyVar2}) ();
5     [] MyPort3.catch(MyProc, ExceptionOne) ()
6         // catch exception
7 }
```

Figure 2.24: Example on Communication Operations

Test Data Specification A test system exchanges data with the SUT. Herefore, test data must be defined. TTCN-3 offers different constructs to describe test data. Data types, templates, variables, procedure signatures can be used to express messages and signatures. Besides, TTCN-3 also offers the possibility to import data described in other languages.

Besides basic data types, e.g. **integer, charstring** or **boolean**, TTCN-3 also supports complex types such as **records, sets** or **unions**. Furthermore, TTCN-3 supports the test-specific types in order to define verdicts, defaults and any kind of data types. The **verdicttype** enumerates the possible verdicts of a test, i.e. **pass, fail, inconc, error** and **none**. The **anytype** is an union of all known TTCN-3 types. It is used as a generic object which

```
1  /** Templates for basic types **/
2  template integer t1 := {1, 2, 3};
3  template charstring t2 := ?;
4
5  /** Data Type and templates for structured types **/
6  type record MyRecordType {
7        integer field1 optional,
8        charstring field2,
9        boolean field3
10 }
11
12 template MyRecordType t3 := {
13       field1 := omit,
14       field2 := "ABC",
15       field3 := false
16 }
```

Figure 2.25: Example on Types and Templates

is evaluated when the value is known. The **default** type is used for default handling and represents a reference to a default operation.

Templates are data structures used to define message patterns for the data sent or received via ports. They are used either to describe distinct values transmitted via ports or to evaluate if a received value matches a template specification. Figure 2.25 shows the definition of the basic template t1 with a value between one and three, and a template t2 of any kind of charstring value. Furthermore, a structured type MyRecordType and its template t3 are specified. The record type owns three fields field1, field2 and field3, where field1 is an optional field. In its template definition, the first field is omitted and the other fields are set by concrete values.

2.3 Summary

In this section, basic knowledge about the MDA Framework has been introduced, including its embraced technologies and languages called MOF 2.0, UML 2.0 and QVT. MOF 2.0 is a meta-metamodeling language which is used to define other languages. In this work, it is needed in order to define a new language called *Test-Directive Metamodel* in Chapter 5. UML 2.0 is a standardized modeling language which serves as a foundation for the test modeling language called U2TP defined in Chapter 4. The QVT standard provides model transformation technologies. In order to achieve the model transformation in this work, a transformation language called *Tefkat* and its underlying engine are used. The realization of the transformation is shown in Chapter 5.

Besides the detailed introduction of the MDA Framework, testing concerns have been illustrated in Section 2.2 of this chapter. Herein, testing is categorized by the dimensions of test scopes, test goals, test distributions and test development phases. The emphasis of this work lays on both the functional and real-time black-box testing. At last, the standardized executable test specification language called TTCN-3 is presented in detail. TTCN-3 has been defined for functional testing. In order to be able to specify real-time tests with TTCN-3, the language is extended by real-time test concepts in Chapter 6.

Chapter 3

System Modeling with UML 2.0 – A Case Study

"You have to see it yourself!" – *Morpheus, The Matrix*

This chapter shows how UML 2.0 can be used to specify system models. As a case study, a roaming technique for data transfer using Bluetooth hardware devices is chosen[1]. After a short introduction about the Bluetooth technology in Section 3.1, the new roaming approach for Bluetooth devices is explained. In Section 3.2, the application is modelled by different UML 2.0 diagrams as a platform-independent system model. This case study will accompany the reader throughout the whole thesis.

3.1 A Roaming Approach for Bluetooth Devices

Bluetooth is an established standard for short-range wireless communication. The Bluetooth specification enables small devices to interact within a short range [Blu04]. These device interactions are known as personal area networks [IBM06c]. The standard embodies both hardware (i.e. radio, baseband and hardware interface) and protocol layers specifications.

In the Bluetooth standard, two network types called *piconet* and *scatternet* are defined. A Bluetooth network consists of master and slave devices. Between the master and its slaves, data can be transferred whereas the master switches the connection rapidly from slave to slave in a round-robin manner. A Bluetooth device may change its role of being either a master or a slave at any time. A master device is able to communicate with up to seven slaves and a Bluetooth network containing eight Bluetooth devices is called piconet. Additionally, the Bluetooth specification allows connecting

[1]The ideas of the Bluetooth Roaming approach originates from the research project [Blu06] where the author of this thesis also participated to the specification of the system model.

two or more piconets together to form a scatternet. Here, special devices are needed which act as bridges by simultaneously being a master in one piconet and a slave in another piconet.

Roaming is a technical term in wireless telecommunication either to extend connection in a location different from the home location [MP92] or a handover for an ongoing connection between base stations within the same network [WLA03]. Roaming is typically performed if a user gets out of the range of his current base station and gets better connection with another one. However, the current Bluetooth standard does not support roaming of devices between its networks, e.g. piconet. There are several reasons why the Bluetooth standard does not support roaming, including:

- The main usages of Bluetooth are cable replacement of locally fixed devices and support of short-term adhoc networks (e.g. synchronization of desktop PCs and PDAs). For these applications, roaming is not required.
- Bluetooth is supposed to be very simple, efficient and cost-saving. Roaming support would enlarge the Bluetooth protocol stack, making devices more expensive.
- It is hard to define the roaming boundary in piconets because every Bluetooth device can be both master and slave.

Therefore, if a Bluetooth slave device is losing the link to its master, no provision is made to switch the connection to another master. Nevertheless, roaming within Bluetooth networks might be useful in some cases. Assuming having more than one Bluetooth LAN access point, roaming might be useful for having a seamless connection even while moving. In the following, a new roaming approach for the Bluetooth technology is described, concentrating on the piconets as the network topology.

3.1.1 The Bluetooth Roaming Application

The needs for a basic roaming support for Bluetooth devices is situated in a medical environment[2]. Its goal is to replace the traditional cable-based monitoring of patients during surgical treatments with a wireless transmission of the patient's monitoring data using Bluetooth hardware devices [PDGN03, DGNP04]. By transmitting the sensor data via radio, the mobility of the patient shall be increased significantly. Also, the number of artifacts caused by cables and the overall cost for the replacement of broken cables can be reduced.

[2]This roaming approach originates from a project between the clinical center and computer science institutes at the University of Luebeck [Blu06].

The idea is as follows: Sensor data like electrocardiogram (ECG), temperature or blood pressure are gathered at a mobile device, digitized and transmitted via radio to fixed units, the receivers. The mobile device is fixed at the patient's bed or the patient itself which may be moved during the entire monitoring period. One of the advantages of this wireless monitoring is a continuous data transmission throughout all the different stages the patient passes through (e.g. preparation → anesthesiology → surgery → wake up → intensive care). Thus, the connection between the mobile devices and the receivers mounted at the hospital's ceilings must be handed over from one receiver to the next while the patient is moving. The receivers must cover the entire area which the mobile device can reach. In order to allow a seamless connection, the areas covered by the antennas of two adjacent receivers must be overlapping (Figure 3.1). Different units (e.g. sensor, digitizing or radio transmission units) share the same rechargeable battery pack. The electric power consumption plays an important role in the design of the system. As a consequence, a mobile device only consists of a small embedded device including the Bluetooth chipset and a low-current microcontroller without a complete Bluetooth protocol stack running on it.

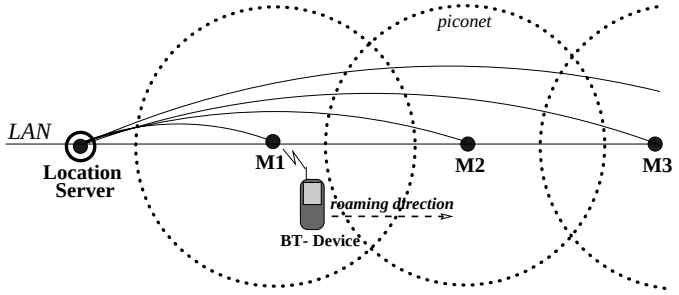


Figure 3.1: Network Configuration

3.1.2 The Roaming Algorithm

Before describing the roaming algorithm, the basic procedures for building up connections between Bluetooth devices[3] are explained first: In order to establish new connections between Bluetooth devices, the Bluetooth procedures called *inquiry* and *paging* are used. The inquiry procedure enables a device to discover which other devices are in its range and what their unique Bluetooth device addresses are. Knowing the correct Bluetooth device address, a connection can be established with the paging procedure.

[3]From now on, a *Bluetooth device* denominates a device using a Bluetooth hardware unit to send and receive data without necessarily using the complete Bluetooth protocol stack.

Assuming the paged device accepts the connection request, both devices receive a connection confirmation message and the connection is established. A device which confirms the connection establishment first will automatically become the master of the connection while the other device becomes the slave. If both devices agree, the roles can be switched at any time after the completion of the connection procedure.

In this roaming approach, all receivers are connected to a fixed network which also connects the receivers to other applications or servers (Figure 3.1) [PDGN03]. The mobile devices are usually moving along the receivers. Because of the piconet structure, the receivers as the central units are assigned to masters and the mobile devices are assigned to slaves by definition. If a slave runs the risk of losing connection to its current master, the connection must be handed over to the next master. By periodically checking the quality of the link[4] to the master [Blu04], the slave is able to detect the connection loss in time. Whenever the quality drops below a certain threshold value the next master will be chosen. Because the inquiry procedure is very time-consuming (up to 10s) and no data transmission is possible during this period, the slave tries to connect directly to the next master using paging (consuming less than 100ms). This implies that the slave knows to which master it has to contact next.

However, the problem is that a slave has no detailed knowledge of its spatial position. For this reason, the information about potential new masters has to be provided by the current master. In order to create a suitable list for each slave, the movements of the slaves are tracked by a *Location Server*. The Location Server stores the physical positions of all receivers e.g. in form of a connection graph. For each slave that is registered within the network, an ordered list of the masters visited before is stored in the Location Server. With this list, prediction can be made to determine which master can be connected to in the next future. So if the slave connects to a master, the Location Server will be informed by the master. The Location Server then builds up a roaming list of potential new masters, and returns it to the master of the slave. After the reception of this list, the master forwards it to the slave. The order of the roaming list depends on the prediction made by the Location Server, i.e. the list contains all masters that may be reached at the next step starting with the master with the highest probability going down to the masters where the probability is very low. After the connection is set up between the slave and a master, data can be exchanged and the link quality be checked periodically.

Figure 3.2 visualizes the roaming algorithm in an UML 2.0 Activity Diagram. The diagram shows the activities of a slave while roaming. In the beginning, the slave tries to connect to a master. If the connection is successful, the

[4]This can be done using the *HCI_Get_Link_Quality* command defined in the Bluetooth standard.

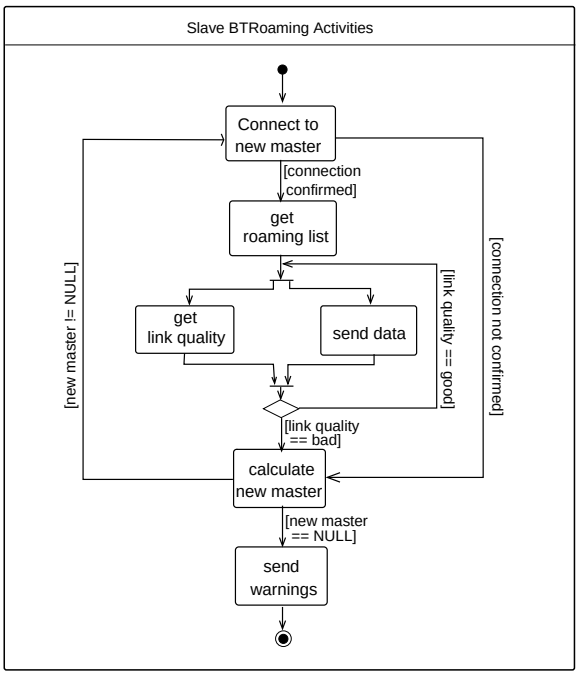


Figure 3.2: Roaming Algorithm as Activity Diagram

updated roaming list is transferred to the slave and data can be sent. In parallel, the link quality between slave and master is observed. If the quality gets bad, the slave will look up in its roaming list for a new master and try to connect to that master directly. If, for any reason, no connection can be established and thus no connection confirmation is received by the slave, a warning message is sent to the user (e.g. by a warning light or a special sound indicating that a problem has occurred). Another warning message is sent to the last master. If the connection to the last master is still alive, the reception of a warning message can be used to initiate appropriate exception handling mechanisms.

Figure 3.3 shows a typical roaming scenario in a UML 2.0 Sequence Diagram. Four instances are involved in his scenario: a Location Server, a Slave and two master instances Master1 and Master2. Slave sends data to Master1 and checks the link quality. If the result is below a predefined threshold value (i.e. $\neq$good), a new master will be calculated by Slave by means of its roaming list. In this scenario, Master2 is the new master. Thus, Slave sends a connection request to Master2 and gets a confirmation indicating that the connection has been successfully established. Afterwards, Master2 informs the Location Server that it is now the new master of Slave. Master2 receives the new roaming list from the Location Server and forwards it to

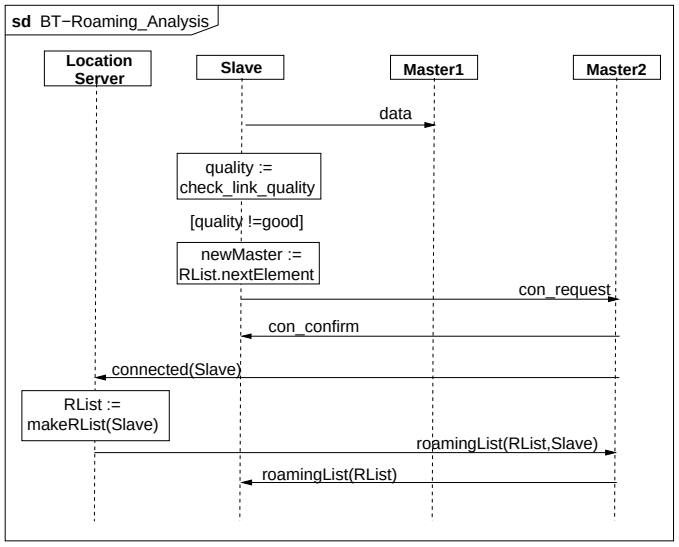


Figure 3.3: Roaming Analysis

Slave. Hence, Master2 can exchange data with the slave until the link quality between them becomes weak again.

3.1.3 Protocol Stack with Roaming Layer

The Bluetooth protocol consists of multiple stacks [Blu04]. For the definition of the roaming approach, the original Bluetooth stack has to be extended. Figure 3.4 shows the new Bluetooth protocol stacks resulting from the proposed roaming approach. Herefore, special protocol layers for roaming called the *Slave Roaming Layer* and the *Master Roaming Layer* are added. These roaming layers take care of the correct connection handover while roaming. Distinction is made between master and slave roaming layers since the roaming functionality depends on the role of the devices. While the slave roaming layer has periodically to check its link quality and calculate new masters before roaming, the master roaming layer has to forward data between its lower and higher layers. Since every Bluetooth device is allowed to be both slave and master according to the Bluetooth specification, the roaming functionalities for both slave and master are located in the same protocol layer[5].

This given Bluetooth roaming approach makes no use of the higher protocol stacks of Bluetooth. Therefore, the roaming layers are implemented directly

[5]In this figure, we separate and distinguish between slave and master roaming layers in order to distinguish the roles of the devices and for a better explanation of the protocol stacks.

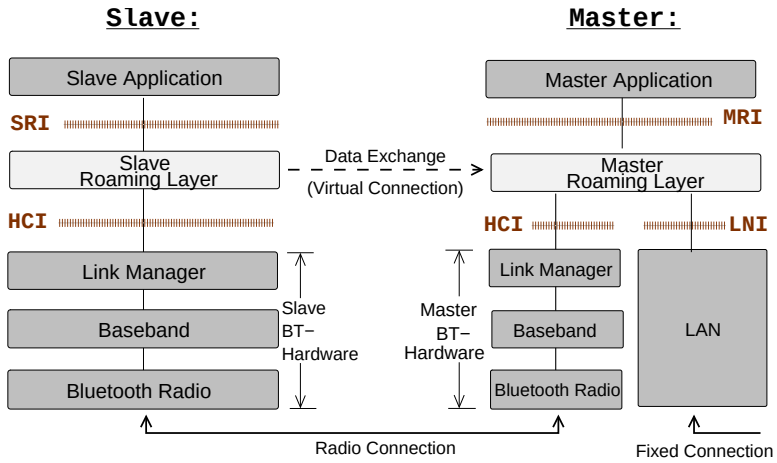


Figure 3.4: Protocol Stack with Roaming Layer

on the hardware interface called *Host Controller Interface* (HCI). The Bluetooth application layers are set upon the roaming layers. The interface from roaming layer to application layer is called *Slave Roaming Interface* (SRI) and *Master Roaming Interface* (MRI), respectively. Moreover, a master is specified as a fixed network node. Thus, it also embodies the LAN protocol stacks to be able to communicate with the local network. The interface between the Master Roaming Layer and the Ethernet is called *Local Network Interface* (LNI).

3.2 Modeling the Bluetooth Roaming Application

In the following, the roaming approach is modeled by various UML 2.0 diagrams types for both architecture and behavior of the system.

3.2.1 System Architecture

Figure 3.5 shows a UML 2.0 package with different classes involved in the Bluetooth roaming approach. Similarity can be recognized between the classes in Figure 3.5 and the Bluetooth protocol stacks in Figure 3.4 where the lower layers of the roaming layers are grouped together, illustrated as a single class. Additionally, a class for the location server is specified.

The slave classes are called Slave Application, Slave BTRoaming and Slave BT-HW (Bluetooth Hardware). The interfaces SRI and HCI connect the classes with each other. A Slave BT-HW is connected to one Master BT-HW. Similar to the slave classes and interfaces, there are the classes Master Application,

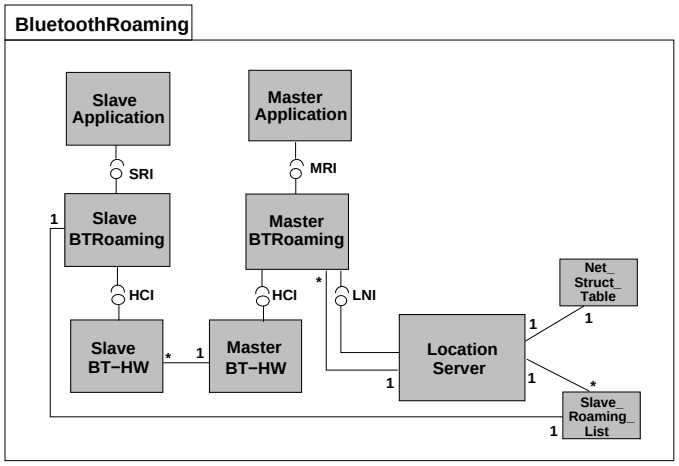


Figure 3.5: Bluetooth Roaming Package

Master BTRoaming and Master BT-HW and the interfaces MRI and HCI on the master's side. Furthermore, the Master BTRoaming class is connected to the Location Server, which by means of the interface LNI represents a node in the local network. The Location Server owns a Net_Struct_Table and several Slave_Roaming_Lists. For each slave, there is exactly one roaming list. The Net_Struct_Table is a static table which provides information about the structure of the local network and the physical position of the masters as necessary for calculating each Slave_Roaming_List. In contrast, the Slave_Roaming_List is changing dynamically. It is updated by the Location Server whenever a slave roams to a new master. Addtionally, a copy of each updated Slave_Roaming_List is transferred to the slave.

Figure 3.6 shows the configuration of a Bluetooth network in a UML 2.0 Composite Structure Diagram. This network consists of one slave, two masters and a location server, including their application, roaming and hardware parts for the slave and master devices[6]. The classes of the parts rely on the classes shown in Figure 3.5[7]. In this diagram, the ports of the parts are also specified. Ports are depicted as small rectangles attached to the part, e.g. part sr of class Slave BTRoaming has two ports called p_sa connecting to port p_br of part sa, and the second port called p_hw connecting to port p_s of part sh.

[6] A *part* in a UML 2.0 Composite Structure Diagram declares that an instance of an classifier may contain a set of instances by composition.

[7] The Location Server, Net_Struct_Table and Slave_Roaming_List are disregarded in the diagram for space reasons.

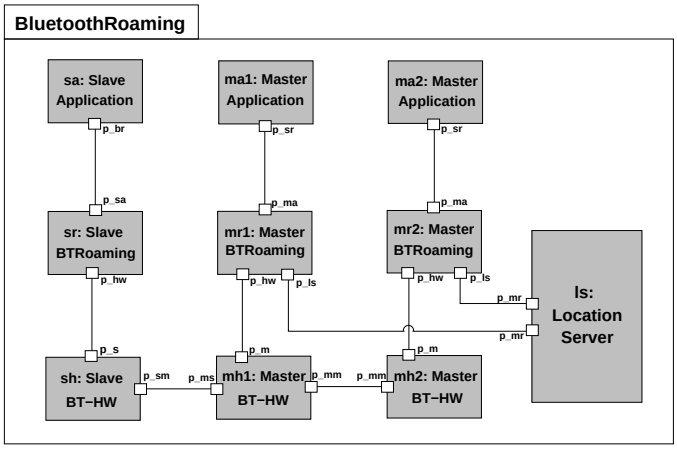


Figure 3.6: System Configuration in Composite Structure Diagram

3.2.2 System Behavior

A detailed system scenario of the Bluetooth roaming approach can be found as an Interaction Diagram in Figure 3.7. Eight instances are specified in this diagram: One instance of class Location Server called ls, three slave instances of class Slave Application, Slave BTRoaming, Slave BT-HW, and four master instances of the classes Master BT-HW and Master BTRoaming for the two masters.[8]

The scenario starts with a connection request from the application instance of the Slave to Master1.[9] The hardware instance mh1 confirms the connection establishment and the roaming instance mr1 informs the Location Server that the Slave is now under its responsibility. Hence, the Location Server calculates and updates the roaming list RList of the Slave and sends it to mr1. mr1 forwards the RList immediately to the roaming instance of the Slave sr. Henceforward, data can be exchanged between the Slave and Master1 until the link quality becomes bad.

The verification of the link quality is done periodically every 0.5 seconds between the roaming instance sr and the hardware instance of the Slave sh. If the link quality is proved to be bad, a new master is needed. For that, the function Search_NewMaster() is called by the roaming instance of the Slave. This function looks up in the RList and picks out the name of the best reachable neighbouring master and returns the name of the new Master to sr instance (Figure 3.8). In case that RList is empty, a warning will be

[8]The application instances of the masters are not shown because roaming is independent from the application layers.

[9]In order to provide an intuitive understanding of the signals, the names are abstracted from the real Bluetooth signal names in the Bluetooth specification [Blu04].

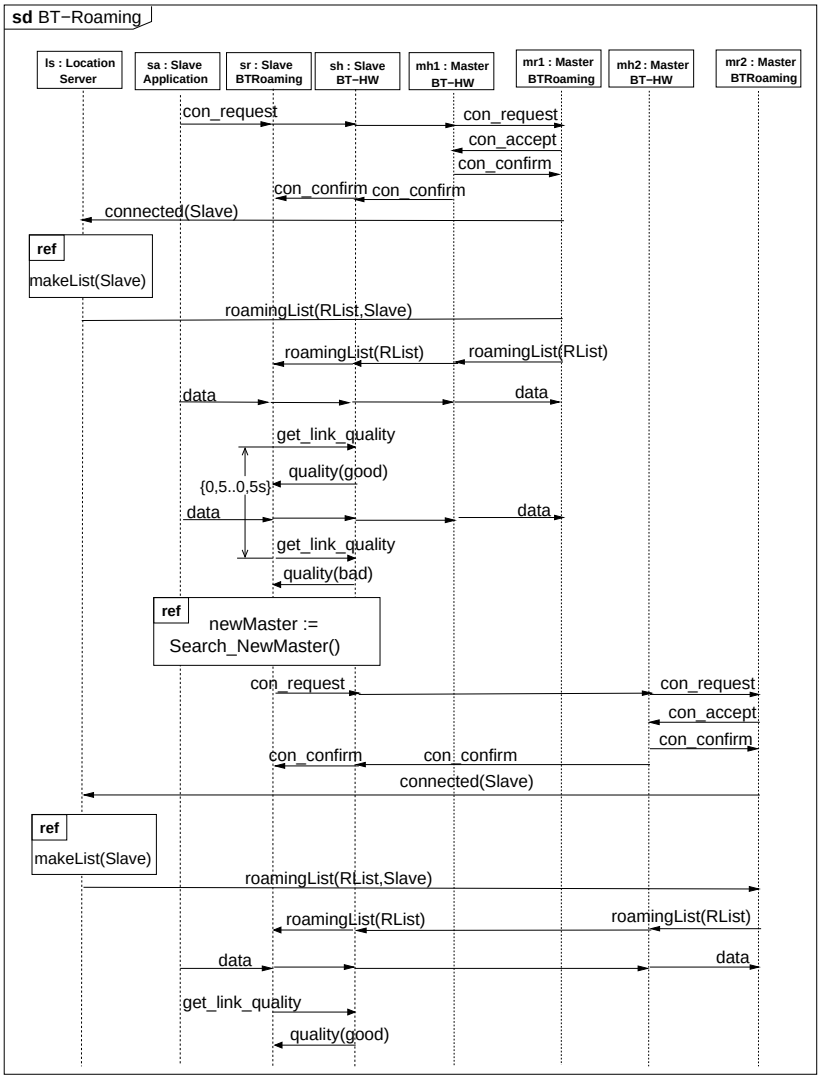


Figure 3.7: Roaming Scenario Design

sent to both the application instance of the Slave sa and the old master (if it is still possible). Additionally, a NULL value is returned to the roaming instance of the Slave.

In the scenario shown in Figure 3.7, Master2 is chosen to be the new master. Thus, a connection request is sent from the roaming instance of the Slave to the roaming instance of Master2. If the hardware instance of Master2 mh2 confirms a successful connection establishment, the Location Server will again be informed about the new status of the Slave. It updates the roaming list and sends it to the roaming instance of the new master. The

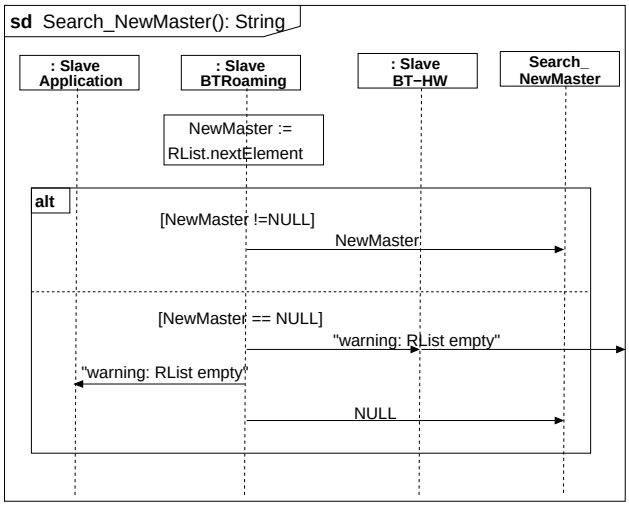


Figure 3.8: Search_NewMaster() Function

roaming instance of Master2 mr2 forwards the list to the roaming instance of the Slave and data exchanges can be started.

For a more detailed local view on a single class, UML 2.0 State Machine Diagram provides good means. Figure 3.9 shows a State Machine of the roaming instance of a Bluetooth slave device which receives messages from its application and hardware instances. This State Machine contains three main states Disconnected, Connected and Roaming. Disconnected is a composite state with multiple sub-states and Connected has the orthogonal sub-states Connect and Check_Link_Quality.

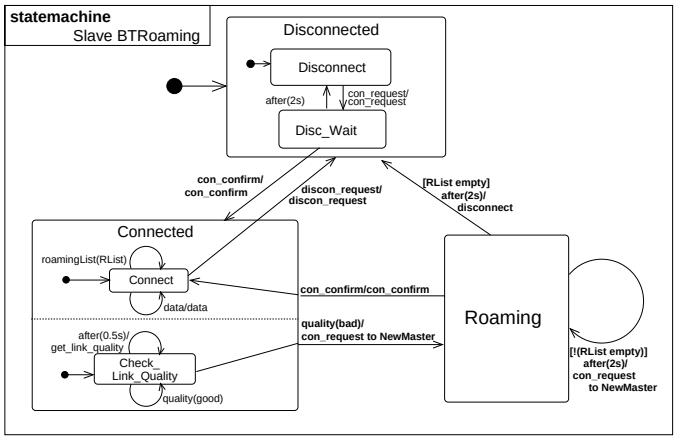


Figure 3.9: State Machine of class Slave BTRoaming

In the beginning, the roaming instance of the Slave is in the sub-state Disconnect of state Disconnected. If it receives a connection request from the application instance, it forwards the request to the hardware instance and goes into the sub-state Disc_Wait, waiting for a connection confirmation from the hardware. If the confirmation message does not arrive within 2 seconds, Slave BTRoaming instance goes back to the Disconnect state. If the confirmation is received by the roaming instance within time limit, the slave is Connected and goes into the sub-state Connect. In this state, data can be received from the application instance and forwarded to the hardware instance (state Connect). In parallel, the link quality is verified periodically every 0.5 seconds (state Check_Link_Quality). The Roaming state will be reached, if the link quality becomes bad. Herein, a new master is picked out and a connection between the slave and the new master will be established. From state Roaming, the roaming instance can either get Connected to a new Master or be Disconnected again, if the roaming list has been exhaustively searched and no master can be found.

3.3 Summary

In this chapter, a case study has been introduced where UML 2.0 is utilized to specify a system model for a potential roaming technique for data transfer scenarios using Bluetooth hardware devices. For the design of the system model, different UML 2.0 diagram types have been taken to model the various aspects on structure and behavior of the system. This platform-independent system model serves as an initial model for the test model specification in the upcoming chapters.

Chapter 4

Test Modeling with U2TP

"To model or to program, this is not the question." – *Anonymous*

UML 2.0 is a great improvement of its previous versions. In particular, its generic extension mechanism supporting profile definition for different domains is promising to offer a common ground for both system and test modeling. However, since UML 2.0 still suffers from missing concepts for describing test models efficiently, the OMG defined a UML 2.0 Testing Profile (U2TP) to support test model specification.

In the MDT approach defined in this work, U2TP is utilized as the test modeling language before test code is generated from a test model. This chapter gives a detailed insight into the profile and its concept definitions. After some historical background knowledge about the profile, the definition of the U2TP concepts will be discussed in detail. Its applicability is shown for both functional and rea-time test modeling by specifying a test model for the Bluetooth roaming algorithm in Section 4.4. To do so, the system model specified in Chapter 3 is taken into account.

The standardization work on U2TP has been done by a consortium. The author of this thesis is a member of the U2TP consortium and actively participated in the standardization work of the profile. The emphasis of her contribution to the profile laid on the definition of the test behavior concepts, the time concepts and the mapping between U2TP and TTCN-3. A paper submitted by the complete U2TP consortium can be found in [BDG+04].

4.1 The Genesis of U2TP

Often, testing is not well integrated with other development phases in the software development process. One reason for this is that designers, developers and testers all use different languages and tools, making it difficult to communicate with each other and to exchange documents.

A UML 2.0 model primarily focuses on the specification of the structure and behavior of a system. However, UML 2.0 provides barely means to specify test artifacts. Hence in 2001, the OMG issued a Request For Proposal (RFP) to develop a testing profile for UML 2.0 [OMG02b]. By means of such a profile, UML 2.0 should be extended by test domain specific concepts to enable extensive test model specification for black-box testing.

Seven organizations consisting of tool vendors, industry users and research institutes[1] responded to the call of the OMG. Early in the process, these organizations decided to collaborate and produce a joint submission. The U2TP consortium is built of a group of both testers and modelers. While some of the consortium members come from the world of testing, e.g. unit testing, system testing and telecommunication testing, others are experienced in system modeling with MSC, SDL and UML, respectively. These members agreed that by working in a larger team, the resultant profile should have a broader scope on black-box testing, where the best ideas should be incorporated into the profile specification.

After two years' work, a testing profile named the UML 2.0 Testing Profile has finally been reviewed and approved by the OMG. Since 2004, U2TP has become an official OMG standard. The profile is developed in the follwing phases:

1. The concept space combining various testing areas is defined.
2. Test concepts are realized by stereotype definitions extending UML 2.0 meta-classes.
3. Mappings to existing test infrastructures demonstrate the practical use of the profile.

In the following, the definition of the test concept space and its realization in the profile are provided in detail. Mapping to test infrastructures is explained in Chapter 6.

4.2 Introducing U2TP Concepts

U2TP provides concepts which target the development of test models for black-box testing where the internal events of the SUT are kept hidden [U2T04]. The profile is based on UML 2.0 and extends its meta-model by the stereotype extensibility mechanism. The benefit of defining a UML 2.0 profile is to allow reuse of existing UML 2.0 concepts and thus minimize the introduction of new concepts.

[1] The U2TP consortium is formed by Ericsson, Fraunhofer FOKUS, IBM/Rational, Motorola, Telelogic and the University of Goettingen.

The U2TP document introduces four concept groups covering the following aspects: test architecture, test behavior, test data and time [Dai04b]. The main concepts can be found in Table 4.1. Together, these concepts define a modeling language to visualize, specify, analyze, construct and document a test system. When selecting and defining the U2TP concepts, test concepts recommended by the OMG, ETSI and ITU-T [OMG06, ISO97] provide a valuable guideline to the consortium. Above that, test languages such as JUnit or TTCN-3 also proved themselves to be valuable [JUn06a, ETS05a]. In the following, the four groups of concepts are introduced in detail. Their concrete profile definition is described in Section 4.3.

Test Architecture Concepts	Test Behavior Concepts	Test Data Concepts	Time Concepts
SUT	Test objective	Data pool	Timer
Test component	Test case	Data partition	Time zone
Test context	Defaults	Data selector	
Test configuration	Verdicts	Wildcards	
Test control	Validation action	Coding rules	
Arbiter	Test log		
Scheduler			

Table 4.1: Main U2TP Concepts

4.2.1 U2TP Test Architecture Concepts

The test architecture group contains the concepts needed to describe elements involved in the test and their relationships. For the specification of the test architecture, any kind of UML 2.0 structure diagrams can be used, e.g. Class Diagram or Package Diagram.

Figure 4.1 shows a general schema for black-box testing. In it, the `System Under Test (SUT)` with all its objects to be tested is illustrated as a black box. It can only be accessed by operations via public available interfaces. The test system consisting of `test components` interacts with the SUT. Communication connections among test components and to the SUT must be configurated before tests are executed. In U2TP, a test component is an active object within a test system performing the test behavior.

Tests are performed in a certain context. The `test context` concept in U2TP is defined more generally than merely as the environment for programm execution. It consists of test information like test cases that belongs to the behavior aspect of testing, test configuration which reflects the test environment, as well as test control. In a `test configuration`, the initial setup of the test components and the connection to the SUT is specified. All test cases listed in the same test context share the same test configuration.

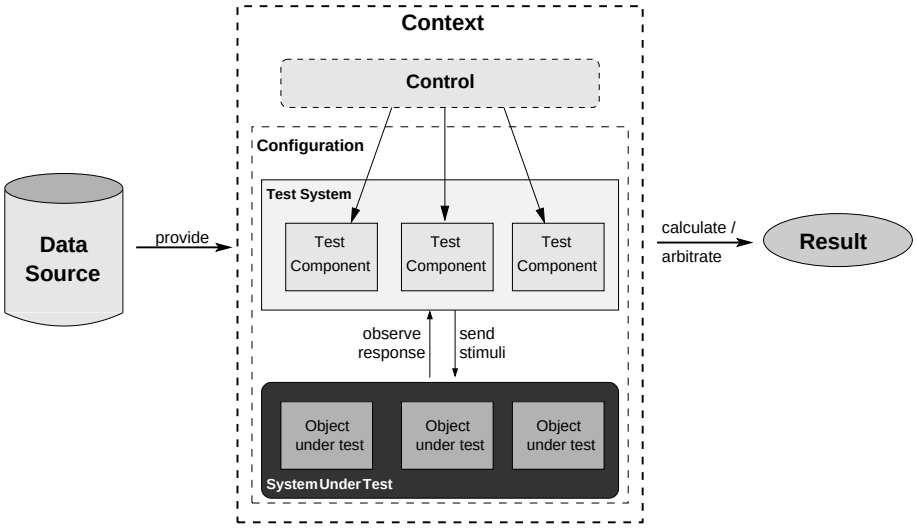


Figure 4.1: General Schema of Black-Box Testing

The configuration may change during test run when components are dynamically created or destroyed. A `test control` specifies the invocation of the test component behavior within a test context. In U2TP, the behavior of a test component is specified in test cases. Above that, decisions about test case executions are made in the test control specification. E.g. in case a test case fails, the whole test could be stopped and failures in the system can be fixed before executing further test cases in order to save time and resources.

A test must deliver its results in a certain form to the external world. In case of U2TP, they are in form of `verdicts` and the evaluation of the verdicts is performed by a denoted component in the test system called `arbiter` to calculate the final test result. Arbitration is an evaluation process to calculate the final test result derived from local results of the test components in a test system. To do so, arbitration strategy must be provided by the arbiter interface. In U2TP, the arbiter provides interface operations in order to get and set the test verdict. To get a verdict, a test component provides its temporary local verdict to the arbiter. To set a verdict, the arbiter is invoked by a validation action and the test verdict stored in the arbiter is updated. According to U2TP, every test context should have an implementation of the arbiter interface. If there is no specific arbiter interface implemented, a default arbiter like the one recommended in the U2TP specification must be taken into account. This default arbiter of U2TP is based on the verdict overwriting rules defined in TTCN-3 (Section 2.2.3). Herein, the arbiter only allows that the final test verdict becomes worse, i.e. a **pass** verdict may be overwritten by a **inconc** or a **fail** verdict.

In addition, a `scheduler` is needed to control the creation and destruction of the test components since test components can be created and destroyed at any time during test execution. It also maintains the existence and is aware of the participation of each test components in all test cases. A scheduler collaborates with the arbiter and informs it when to issue the final verdict. Further features concerning verdicts are dealt with in the U2TP test behavior concepts.

The SUT is stimulated and observed by the test system which gets its test data from some data sources which may be a pool of data, partitions of data with disjoint data-sets or real-time data generated by sensors or detectors. In U2TP this part is specially dealt with and refined in the U2TP test data concepts.

For a distributed test system, test components are assigned to time zones in order to achieve time-synchronization among themselves. This is treated in the U2TP time concepts.

4.2.2 U2TP Test Behavior Concepts

Test behavior is the concrete technical specification of the dynamic behavior of a test using e.g. sequences, alternatives, loops and defaults to stimulate and observe the SUT. The test behavior group defined in the U2TP document includes concepts to specify test behavior related to a test context. For the definition of test behavior, any kind of UML 2.0 behavior diagrams may be used, e.g. Sequence Diagram or State Machine Diagram.

In order to express the intention of a test, a `test objective` with a general description of what should be tested is given. Its concrete specification is a `test case`. A test case is a specification of one specific case to test the system, including the required test behavior with its test inputs, test conditions and test result. In a U2TP model, a test case is listed as a public operation in the test context returning a test verdict as its result.

Typically, a test case specification describes the normative or expected behavior for the SUT. To catch unexpected responses such as exceptions, default behavior is needed in order to keep the number of erroneous test case executions to a minimum. A `default` specification is a means to make partial behavior definitions of test components complete and robust against unexpected behavior of the SUT. Predefined default behavior provide certain convenience to the test developer. In U2TP, a default is activated by applying it to static behavioral structures: In a State Machine, it can be attached to a state machine, a state or a region. In an Activty Diagram, a default can be specified for an action or an activity. In an Interaction Overview Digram, defaults can be applied to an interaction fragment. However, if defaults are specified on different behavior levels, e.g. on a state, interaction, instance etc., the evaluation order is the follows: If the inner-

most default fails to recognize the observed behavior, the default of the next level is tried.

Depending on the conformance of the SUT with respect to the sequence of performed events, the behavior description of a test case consists of verdict assignments as its return value [WG99]. Additionally to the ordinary **pass** and **fail** verdicts, the ISO9646 defines further verdicts called *inconclusive* and **error** [ISO97]. In U2TP, each test case returns a `verdict`. Predefined verdicts **pass, fail, inconclusive** and **error**[2] exist: The **pass** verdict indicates that the SUT behaves correctly for the specified test behavior. The **fail** verdict describes that the test case has violated. The **inconclusive** verdict is used when the test neither passes nor fails, but still valid according to the specification of the SUT. Finally, an **error** verdict indicates exceptions within the test system itself.

In U2TP, a tester is allowed to define its own verdicts and arbitration rules. The benefit to do so is to broaden the results of the test evaluation according to the needs, e.g. for real-time tests, special verdicts can be defined according to the time restrictions, such as "time<5ms => pass" meaning that if the measured time value is less than 5 ms, then the verdict is assigned to **pass**. In order to trigger and to inform the arbiter about the test results, a `validation action` is performed by the test component.

Often, test should be evaluated during or after a test run. To do so, test events need to be logged during test execution. With U2TP, test events can be logged for a test context or a test case. These traces visualize test behavior during test execution. They can be recorded in a `test log` and become part of the test specification. The format of a log is not specified by U2TP and is therefore to be defined by the user.

4.2.3 U2TP Test Data Concepts

Test data play an important role during test execution in order to explore all aspects of the SUT. Test data are inputs and outputs exchanged between the test system and the SUT. Thus, concepts to model extensive test data are provided by U2TP for data specification and handling.

During test, the SUT must be triggered repeatedly by different data values. In order to observe its responses, equivalence classes building a set of allowed data values are needed. Typically, these values are taken from a partition of test data or a list of explicit values. In U2TP, a pool of data containing either concrete data values or a partition of data provides good means to associate data sets with test contexts and test cases. A `data partition` defines an equivalence class to denote explicit partitioning of data. `Data`

[2]The definition of the verdicts originates from the OSI Conformance Testing Methodology and Framework [ISO97] and are similar to the verdicts defined in TTCN-3.

`selectors` are operations to retrieve specific test data from a `data pool` or a data partition.

In order to handle test data properly, data matching mechanisms are used to proof the correctness of the responses of the SUT with the system requirements. U2TP provides the well-known `wildcards` *"*"* and *"?"*. Wildcard *"*"* matches to any or null value and *"?"* matches to any value in test data. However, wildcards are especially required for data reception, where unexpected events or events containing different values than specified in the test behavior can be handled. Therefore, they often can be found in default specifications where unexpected behavior caused by unknown data reception is catched.

Every system defines its own data format by encoding and decoding those by rules, such as for IDL for Corba or ASN.1 for telecommunication systems. Specifying `coding rules` appoints a common data format for communication and is usually realized at the system interfaces. Coding rules specified in a test system are basically applied to data value specification to denote the concrete encoding and decoding for these values during test execution. In U2TP, this is done by referencing to the appropriate coding rule within the test model.

4.2.4 U2TP Time Concepts

In the existing UML 2.0 standard [UML05], simple time concepts exists in order to read the current time value or observe time duration (Section 2.1.2). However, for test specification, further time concepts are needed to influence the test behavior. U2TP defines two new time concepts, namely `timer` and `time zone`.

`Timers` provide means to observe and control test behavior. Also, a timer can be used to prevent from deadlocks, starvation and instable system behavior during test execution. In a test case specification, a timer may assure that a test terminates properly. Such a behavior can be controlled by a timer started at the beginning of a test component behavior. Herein, the duration of the timer equals to the expected overall execution time of the test component. Such a timer is called *test component timer*. Timers may also used to control the test behavior with regard to a certain response expected after its triggering. To do so, a *guarding timer* is started after a event triggering and is stopped when the response arrives within the predefined duration. A timer may also postpone the execution of an event by a *delaying timer* [Koc01]. For that, a timer is started followed immediately by its timeout just before the dedicated event is executed. Common uses for a delaying timer are, e.g. that the SUT requires time to get into a state to receive the next signal, or the test behavior is delayed on purpose in order to test the reaction of the SUT, or the system specification determines that the SUT should not send a certain message within a given amount of time.

Figure 4.2 depicts the specification of a test component timer T_tc, a guarding timer T_g and a delaying timer T_d at the test component of type ClassA. T_tc of 6 seconds is started at the beginning of the test component behavior and is stopped before the result of the test is set to pass. The delaying timer T_d delays the triggering of message3 by 0.5 seconds. The guarding timer T_g assures that the message4 is received after the SUT has been triggered by message3.

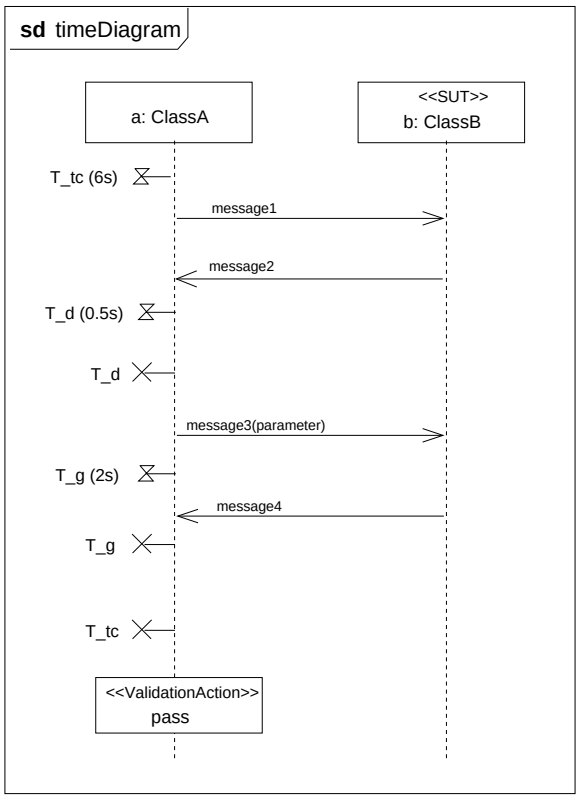


Figure 4.2: Time Concepts

In U2TP, a `timer` is owned by a test component and is started with a positive time value. A timer may be stopped as long as it is still active. The expiration time of an active timer can be read and its status be retrieved. When a timer expires after its predefined duration, a timeout is generated automatically and is sent immediately to its owned test component.

In a distributed test system, multiple test components may exist that are spread over different locations belonging to different time zones. In order to guarantee proper communication within the test components, they must be synchronized. In U2TP, the `timezone` concept serves as a grouping

mechanism for test components within a distributed test system to ease potential time value comparison. Each test component belongs at most to one time zone. Test components of the same time zone have the same time value and thus are considered to be time-synchronized.

4.3 Defining U2TP Concepts

After having introduced the concepts of U2TP, their concrete technical definitions realized by the UML 2.0 profiling mechanism are explained in this section. Before doing so, the UML 2.0 meta-classes utilized in the profile definition are listed.

4.3.1 Utilized UML 2.0 Meta-Classes

U2TP extends the UML 2.0 meta-model when defining its test concepts. Below is a list of all UML 2.0 meta-classes used to define the U2TP concepts. Since the meta-model is structured in packages according to the UML 2.0 standard, the packages of the meta-classes are also mentioned to avoid ambiguity [UML05]. The detailed meaning of the meta-classes will be explained inline the U2TP concept definitions in the succeeding subsections.

UML 2.0 Meta-Class	*Package*
Property	InternalStructures
Classifier	Kernel
StructuredClassifer	InternalStructures
Dependency	Dependencies
NameSpace	Kernel
Behavior	BasicBehaviors
Message	BasicInteractions
Operation	Collaborations
TimeTrigger	Communications
ValueSpecification	Templates
LiteralSpecification	Kernel
Action	CompleteActivities
SendObjectAction	IntermediateActions
CallOperationAction	IntermediateActions
AcceptEventAction	CompleteActions
ReadStructuralFeatureAction	IntermediateActions
WriteStructuralFeatureAction	IntermediateActions

Figure 4.3: Utilized UML 2.0 Meta-Classes for U2TP Definition

4.3.2 Defining Test Architecture Concepts

Figure 4.4 shows the profile definition of the U2TP test architecture concepts. The `SUT` stereotype extends the `Property` meta-class of UML 2.0. The meta-class `Property` from package *InternalStructures* represents a set of instances owned by a classifier instance. When an instance of a classifier is created, a set of instances corresponding to its properties may be created. These instances are instances of the classifier of this property. The stereotype `<<SUT>>` is applied to one or more properties of a classifier to specify that they constitute the SUT. Features and behavior of the SUT are given by the type of the property to which the stereotype is applied.

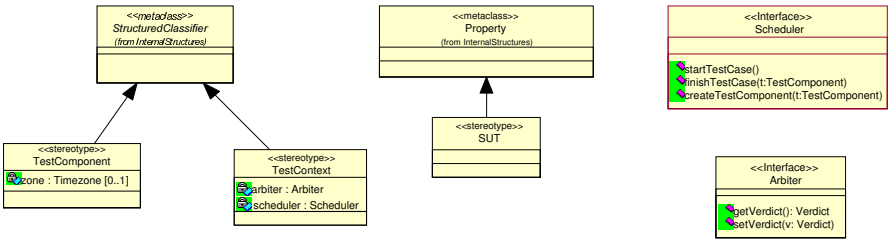


Figure 4.4: Test Architecture Definitions

The `TestComponent` stereotype extends the abstract meta-class `StructuredClassifier` from package *InternalStructures*. The `StructuredClassifier` meta-class represents any classifier whose behavior can be described by a collaboration of instances. A test component is assigned with the `<<TestComponent>>` stereotype and is an active class with a set of ports and interfaces. A test component may own a time zone to specify how it is synchronized.

`TestContext` also extends the `StructuredClassifier` meta-class. The test context is a grouping mechanism for test cases, test configuration and test control. The test context owns an arbiter and an scheduler as attributes. The notion of a test context is `<<TestContext>>`.

In contrast to meta-class extension where operations are not allowed to be defined according to UML 2.0, an interface may define operations within a profile. An `Arbiter` provides operations and is therefore defined as an interface in the profile. It provides the operations *getVerdict()* to retrieve a verdict and *setVerdict()* to determine a test verdict. The verdict setting semantics is defined by the classifier realizing the arbiter interface.

Similar to the Arbiter, a `Scheduler` also provides operations and is defined as an interface. It defines the operations *startTestCase()*, *finishTestCase()* and *createTestComponent()*. When starting the test case, the scheduler will

start the test case by notifying all involved test components. The *finishTestCase()* operation records whenever a test component has finished its execution of this test case. When all test components involved in the current test case have finished, the arbiter will be notified for final verdict calculation.

4.3.3 Defining Test Behavior Concepts

Figure 4.5 and Figure 4.6 show the profile definition of the U2TP test behavior concepts. A test objective is used to specify the aim of a test case and test context, respectively. Stereotype `TestObjective` extends `Dependency` meta-class from package *Dependencies*. The `Dependency` meta-class indicates that a model element requires other model elements for its specification. Thus, the semantics of the depending elements rely on the definition of the supplier elements. A test case or test context can have any number of test objectives. Also, a test objective can be realized by any number of test cases or test contexts. The client of a test objective must be a named element with a test case or test context stereotype applied. The notion of the stereotype is `<<objective>>`.

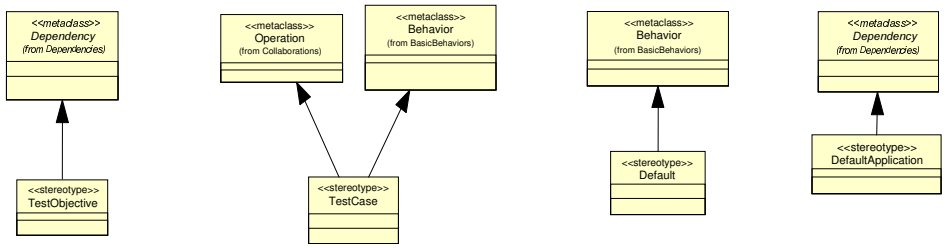


Figure 4.5: Test Behavior Definitions I

The `TestCase` stereotype extends both the `Behavior` and `Operation` meta-classes. `Behavior` is an abstract meta-class from package *BasicBehaviors*. It describes the states a classifier goes through during its lifetime. The `Operation` meta-class from package *Communications* invokes both the execution of method behavior as well as other behavioral responses. A test case is owned by a test context. The notion of this stereotype is `<<TestCase>>`. A test case always returns a verdict.

A `Default` stereotype with the notion of `<<Default>>` extends the `Behavior` meta-class. Defaults may be specified as any behavior diagram, e.g. Interaction Overview Diagram, State Machine or Activity Diagram. For defaults which are described on Interaction Diagrams, an algorithm combining the traces of the default behavior with the traces of the main description exists. This algorithm is described in the U2TP document. The

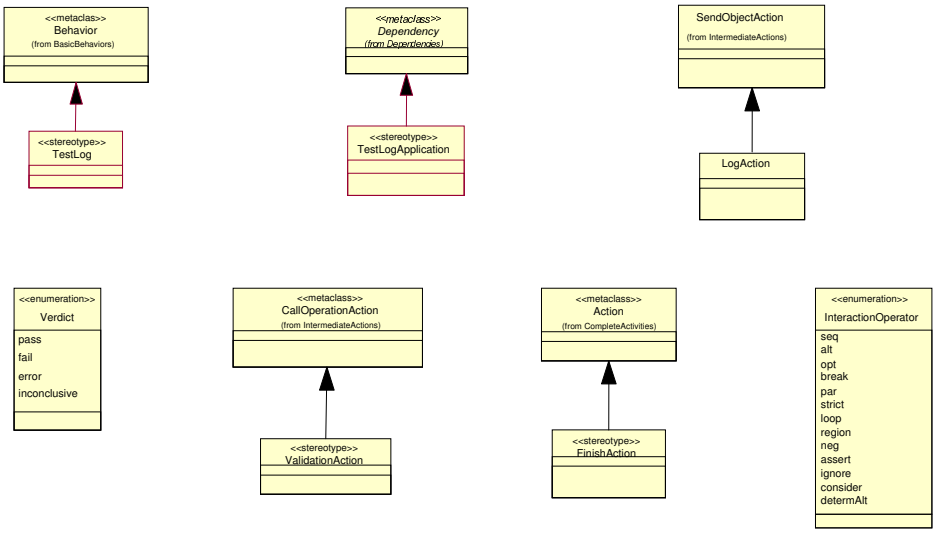


Figure 4.6: Test Behavior Definitions II

`DefaultApplication` extends the `Dependency` meta-class. The notation for a default is identical to a UML 2.0 comment, i.e. a rectangle with a bent corner. The text in the comment symbol has the syntax: **default** default-identifier [**continue — repeat** [repetition-count]]. Figure 4.7a) shows the default attachment of myDefault in a U2TP diagram to the message reception at Lifeline A. According to the U2TP standard, if no statement is given behind the identifier, the behavior is continued by default. Also, if no repetition-count is given, infinity is assumed.

a) Default Symbol b) Test Log Symbol

Figure 4.7: U2TP Default and Test Log Attachment

The `TestLog` stereotype extends the `Behavior` meta-class (Figure 4.6). A test case or a test context may have a number of test logs which represent the behavior resulting from the execution of a test case or a test context. It helps to understand test process which impacts the test case verdict. A test log is applied to a test case or a test context using `TestLogApplication`

which extends the `Dependency` meta-class. The notion of a test log file attached to the test context class myContext can be found in Fiugre 4.7b). Like the `DefaultApplication`, `TestLogApplication` also has the notation of a comment. A `LogAction` extends the `SendObjectAction` meta-class from package *IntermediateActions*. A `SendObjectAction` transmits an object to a target object and invokes a behavior such as a transition within a State Machine Diagram or an execution in an Activity Diagram. The target of a log action refers to a logging mechanism in the run-time system and the request refers to the logged information. The concrete definition for the logging mechanism is not determined by the U2TP standard and is up to the user.

The `ValidationAction` stereotype extends the `CallOperationAction` from package *IntermediateActions*. This meta-class is an action which transmits an operation call request to the target object. Validation actions are used to set verdicts in the test behavior by calling the *setVerdict* operation on the arbiter. The notation for a validation action is an ordinary action with the stereotype `<<ValidationAction>>` applied. If an arbiter is depicted explicitly, a message describing the communication between the test component performing the validation action and the arbiter is shown.

The `FinishAction` stereotype extends the `Action` meta-class from package *CompleteActivities*. This meta-class has a set of incoming and outgoing activities which specify control flow and data flow between nodes. An action only executes when all of its input conditions are satisfied. In an Interaction Diagram, the `FinishAction` is shown as a black rectangle on a lifeline. In a State Machine, it is depicted as a flow branch ending in a black square and in an Activity Diagram, it is shown as a black square.

UML 2.0 defines `InteractionOperators` for Sequence Diagrams and Interaction Overview Diagrams. These operators are typically used to specify certain sequences, alternatives or loops in the system behavior. However, UML 2.0 provides means to define alternatives, where the alternative behavior where the alternative statement becomes valid whenever one of the alternatives satisfies the condition. Sequential evaluation by a given order of behavior cannot be guaranteed. For tests, it is important to have means which determines the order of the evaluation. Thus, U2TP added a new operator to the existing `InteractionOperators`, called `determAlt`, meaning deterministic alternative. With this operator, alternatives can be evaluated sequencially from top to bottom.

4.3.4 Defining Test Data Concepts

Figure 4.8 shows the profile definition of the U2TP test data concepts. The `Data pool` stereotype extends both the `Classifier` and `Property` meta-classes. `Classifier` is an abstract meta-class from package *Kernel* to de-

scribe a set of instances which have common features. A classifier defines a type and owns generalizations, thus enabling the definition of generalization relationships to other classifiers. It can also specify a generalization hierarchy by referencing its general classifiers. Furthermore, it can redefine nested classifiers. A data pool is given by a classifier and the contained data partitions which realizes the data pool. The notation for data pool is a classifier with stereotype `<<DataPool>>`. There are several constraints defined for a data pool: First, a data pool stereotyped classifier can only be referenced by a test context or test component. Second, the stereotyped property can only be applied to a property associated with a test component in combination with a test context stereotyped classifier. Last, a stereotyped classifier cannot be associated with both a test context and a test component.

The `Data partition` stereotype extends the `Classifier` meta-class. The notation for data partition is a classifier with stereotype `<<DataPartition>>`. A data partition stereotyped classifier can only be associated with a data pool or another data partition. The semantics of a data partition are given by the classifier which realizes the data partition. Stereotype `Data selector` extends the `Operation` meta-class. Data selector extends an operation to allow the implementation of different data selection strategies. If a data selector stereotype is applied to an operation, the featuring classifier must either have the data pool or the data partition stereotype applied. The notation for data selector is an operation with stereotype `<<DataSelector>>`.

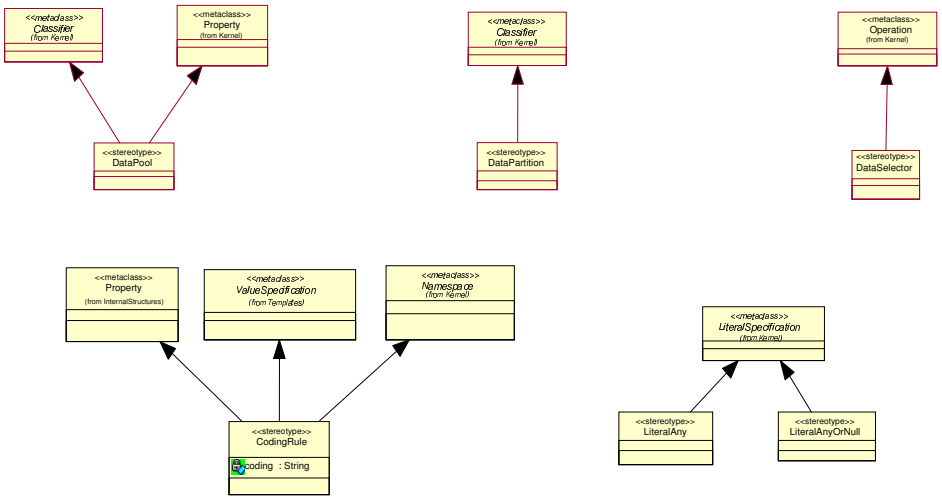


Figure 4.8: Test Data Definitions

`Coding rules` extend the `Property, ValueSpecification` and `NameSpace` meta-classes. The `ValueSpecification` meta-class from package *Templates* specializes parameterable elements. It specifies a value which can be exposed and provided as a formal template parameter. The `NameSpace` meta-class from package *Kernel* is an abstract meta-class. It defines a model element containing a set of elements identified by their names. A coding rule specifies how values are encoded and decoded. If a coding rule is applied to a value specification, it specifies how the values are coded. If it is applied to a namespace, it specifies the coding for all values contained within the namespace. If it is applied to a property, it specifies how the values of this property are encoded. If coding rules are applied to several places in a hierarchy, rules defined at a lower level have precedence over rules defined at a higher level. There is an attribute called `coding` which defines the selected coding scheme. The notation for a coding rule is identical to a UML 2.0 comment. The text in the comment symbol begins with the keyword *coding* followed by the name of the coding rule.

The `LiteralAny` stereotype extends the `LiteralSpecification` meta-class from package *Kernel*. A `LiteralSpecification` identifies a literal constant. `LiteralAny` is illustrated as a wildcard. If it is used as a literal specification, it denotes any possible value out of a set of possible values. If it is used for the reception or a sending of a message. The notation for `LiteralAny` is the question mark (*"?"*). The `LiteralAnyOrNull` stereotype also extends `LiteralSpecification`. `LiteralAnyOrNull` is a wildcard specification representing any value out of a set of possible values, or the lack of a value. If it is used in an Interaction Diagram for a receiving or sending message reception, it specifies that the specified message with any value or without that value is to be received or sent. The notation for `LiteralAnyOrNull` is the asterisk character (*"*"*).

4.3.5 Defining Time Concepts

Figure 4.9 and Figure 4.11 show the profile definition of the U2TP time concepts. `Timer` is defined as an interface with operations to start, stop, read a timer and to check its status. Only an active class may realize a timer interface. Also, a timer may only be started with a predefined positive expiration time. Whenever a timer expires, a timeout is generated automatically and sent to the owner class of the timer. A timer can either be private or public. If private, the timer can only be accessed by its owned active class. If public, anyone may access and manipulate it.

A `StartTimerAction` extends the `CallOperationAction` meta-class. It is an action used to start a timer, triggered by the start operation of the timer interface. The target of the action must refer to a classifier realizing the timer interface. The argument of the action must be a time value.

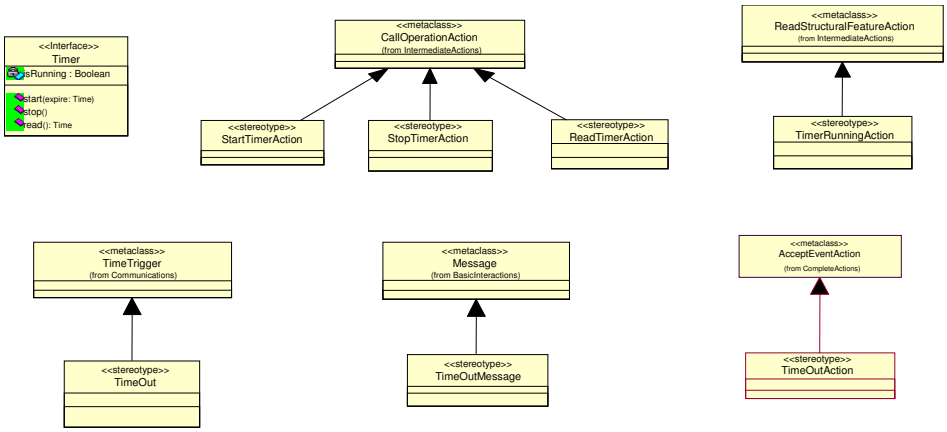


Figure 4.9: Time Definitions – Timer

The notation for this action is an empty hourglass with a solid line connecting with the position where the timer is started. Similiar to stereotype `StartTimerAction`, the `StopTimerAction` stereotype also extends the `CallOperationAction` meta-class. It is an action used to stop a running timer. The target of the action also must refer to a classifier realizing the timer interface. The notation for a `StopTimerAction` is a cross with a thin line connecting the cross and the position where the timer is stopped. Figure 4.10 shows the U2TP timer symbols. `ReadTimerAction` stereotye extends the `CallOperationAction` meta-class as well. It is used to read a timer to obtain its expiration time. The return value of `ReadTimerAction` is a positive time value and **NULL** for a timer which is not running. The notation for a read timer action is an ordinary action with the read timer stereotype applied.

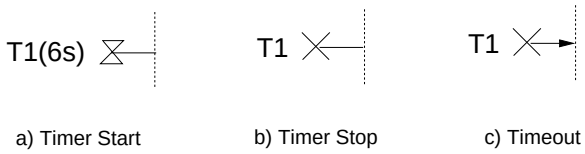


Figure 4.10: U2TP Timer Symbols

The `TimerRunningAction` extends the `ReadStructuralFeatureAction` meta-class from package *IntermediateActions*. `ReadStructuralFeatureAction` meta-class is a structural action which retrieves the values of a structural feature. The `TimerRunningAction` is used to check if a timer is currently running or not. It returns the boolean value **true** if the timer is running,

otherwise **false**. The structural feature of this action is the *isRunning* attribute at the interface of the timer. The target of the action must refer to a classifier realizing the timer interface. Its notation is an ordinary action with the timer running action stereotype applied.

The timeout of a timer is defined for different UML 2.0 diagram types. According to the diagrams, the stereotype for timeout extends different UML 2.0 meta-classes. U2TP specifies three timeout stereotypes in order to cover timeout specification for all UML 2.0 behavior diagrams (Figure 4.9): First, the `TimeOut` stereotype extends the `TimeTrigger` meta-class from package *Communications*. `TimeTrigger` specifies a point in time by an expression which is either absolute or relative to a point in time. A relative time event is used in the context of a trigger. Its starting point is the time at which the trigger becomes active. A timeout can be applied to a Class Diagram or State Machine Diagram. It is generated by a timer when it expires and triggers an associated behavior, e.g. a transition in a statemachine. After a timeout is generated, it is put into the input queue of the owner class of the timer. Further, the `TimeOutMessage` extends the `Message` meta-class from package *BasicInteractions*. The `Message` meta-class defines communication between Lifelines within an Interaction Diagram. A message has a sender and a receiver and can be e.g. a sending of a signal or an invocation of an operation etc. A `TimeOutMessage` is used in an Interaction Diagram and is generated by a timer when it expires. This message is sent to the active class which owns the timer. The notation for the `TimeOutMessage` is an empty hourglass. An arrow with a filled head connects the hourglass and the line where the timeout occurs. Last, the `TimeOutAction` extends the `AcceptEventAction` meta-class from package *CompleteActions*. The `AcceptEventAction` meta-class specified for the Activity Diagram waits for the occurrence of an event meeting specified conditions. A `TimeOutAction` is enabled when a timer expires. The notation for the `TimeOutAction` is again an empty hourglass in an Activity Diagram. An arrow with an unfilled head connects the hourglass and the activity to which the timeout is an input condition.

The `SetTimeZoneAction` extends the meta-class `WriteStructuralFeature-Action` from package *IntermediateActions* (Figure 4.11). With WriteStructuralFeatureAction, the values of an object can be modified. `SetTimeZone-`

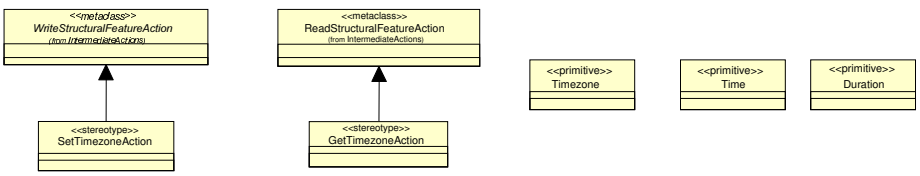


Figure 4.11: Time Definitions – Timezone

`Action` may dynamically set the time zone of a test component. The action can be invoked at run-time by a test component to set its current time zone. The notation for a set time zone action is an ordinary action with the set time zone action stereotype applied.

The `GetTimeZoneAction` extends the `ReadStructuralFeatureAction` metaclass to dynamically retrieve the time zone of a test component. The type of the structural feature of the action and its result must be of type `Timezone`. This action can be invoked by a test component at run-time in order to retrieve its current time zone. The notation for a get time zone action is an ordinary action with the get time zone action stereotype applied.

U2TP defines three primitive types: `Timezone`, `Time` and `Duration`. `Timezone` is used to group test components. Test components with the same time zone value constitute a group and are considered to be time-synchronized. The realization of synchronization is not specified in U2TP. Time zone values can be compared for equality. `Time` specifies concrete time values which are used to express time constraints and to set timers. Statements like now are needed to represent the current time in a system. The difference between two time values is called `Duration`. Durations are used in time constraints and together with time values.

4.4 Applying U2TP to the Bluetooth Roaming Application

The U2TP provides useful concepts for test modeling. Herein, thorough investigation must be made to assure that the SUT is tested accurately. This section introduces a test model for the Bluetooth roaming case study and illustrates the applicability of U2TP for test modeling [DGNP03, DGNP04].

4.4.1 Test Preparation

Before start modeling the test system, test requirements have to be defined. To do so, the system model of the Bluetooth roaming approach introduced in Section 3 is studied carefully. In particular, the aim of the test must be defined, i.e. the test architecture be defined, the SUT and test components be chosen, the test objectives selected.

When defining the test architecture for this case study, two criterias have been considered where decisions and compromises have been made: First, since mature Bluetooth hardware implementations already exist on the commercial market, it is important that real Bluetooth hardware can be taken for testing[3]. Secondly, a flexible test architecture shall be constructed, where the entire network can be extended or reduced by needs.

[3] Usually, test component behavior is emulated.

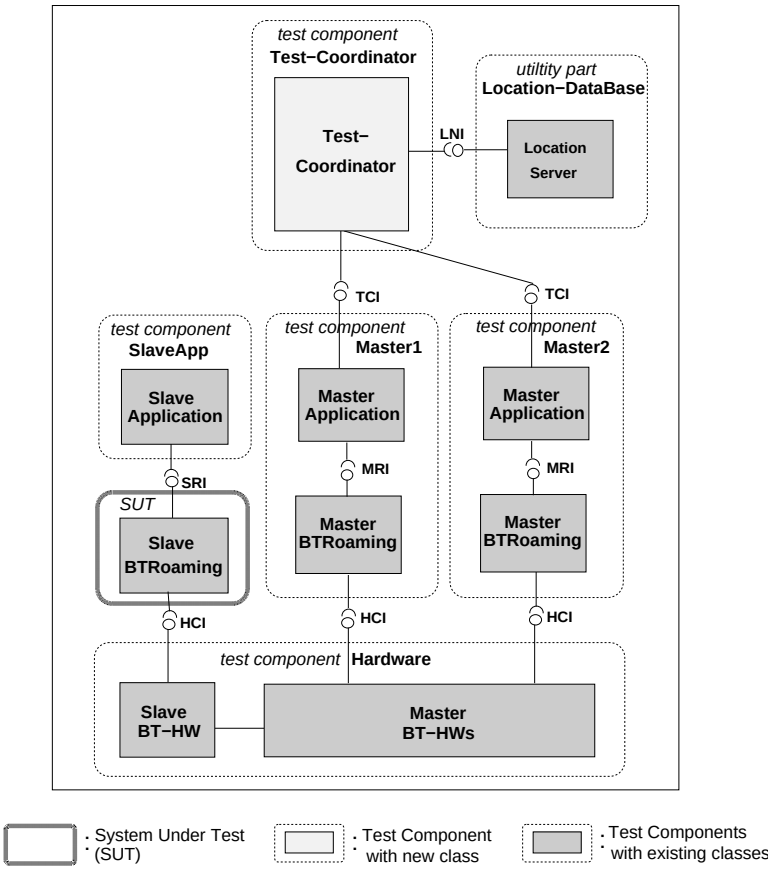


Figure 4.12: Test Architecture with one Slave and two Masters

Having these two criterias in mind, the test architecture for the case study is determined. First, the SUT has to be chosen. Since roaming is the new feature which has been added to the existing sophisticated Bluetooth protocol stack, the correct behavior of the roaming layer is of interest. Moreover, the roaming functionalities performed by the slave is complicated and has therefore to be verified properly. Hence, the roaming layer of the slave is assigned to SUT. Figure 4.12[4] presents a test architecture with one slave and two masters. The classes in the diagram originate from the BluetoothRoaming package of the system model in Section 3. Next, test components in the test system are needed to trigger and observe the SUT during test execution. Slave-Application is a component set upon Slave BTRoaming. It is therefore assigned to test component to enable communication with the SUT. Other

[4]This diagram is not a U2TP diagram. The U2TP model with its test configuration diagram is shown in the successive section.

test components include the underlying Bluetooth Hardware layer containing the hardware layers for both slave and master. The master components Master1 and Master2 embracing the Master Application and Master BTRoaming layers also become test components.

In addition to the slave and master components, there is a new test component called Test-Coordinator added to the presented test architecture in Figure 4.12. It is set upon the master devices. This test component is the main test component in our test system which administrates and instructs the other test components during test execution. Test-Coordinator is also responsible for the evaluation of the test cases and the setting of verdicts during test execution. It also has access to the Location-DataBase which embodies the Location Server and owns the slave roaming lists and the network structure table. Communication between the Test-Coordinator and the masters is performed via an additional Test Coordination Interface (TCI).

With this test architecture, the criterias mentioned for real implementation application and network flexibility are achieved: The Bluetooth hardware can either be emulated by software or with real implementation without influencing the functionality of other components. Moreover, by splitting the Bluetooth hardware from the applications, further slave and master components can easily be added to or removed from the hardware, no matter if the hardware is an emulation or a real implementation.

In order to test the roaming feature, the following functionalities of the roaming layer are of importance and are focus of a test:

- The Slave_Roaming layer should be able to choose a new master by looking up in its roaming list when the connection with its current master gets weak.
- The Slave_Roaming layer should request a connection establishment after a new master is chosen.
- The Slave_Roaming layer should receive a connection confirmation from its new master after a connection between them has been established.
- The connection request is repeated by the Slave_Roaming layer within a certain time if connection is not confirmed by the new master. In this case, another master is asked for connection.
- The Slave_Roaming layer should send a warning to the environment if no new master can be found on the roaming list.
- Both the Master_Roaming and Slave_Roaming layers should be able to forward data between the hardware and application layers properly.

In the following, test architecture and test behavior of the Bluetooth roaming approach is modeled by U2TP.

4.4.2 Modeling Test Architecture

When start modeling with U2TP, a test package containing the test model has to be created first. For the Bluetooth case study, the test package is called Test_BluetoothRoaming. Figure 4.13 shows the Package Diagram of the Bluetooth Roaming test model. This test package imports classes and interfaces from the system model specified in the BluetoothRoaming package in order to have access during the test. In the test preparation phase introduced before, the Slave BTRoaming layer is assigned to SUT and other components are grouped to diverse test components. According to this, the test package consists of five test component classes. The SUT is not modeled explicitely in the test package because the SUT is not a component in the test system and can therefore only be associated with interfaces and ports defined by test components.

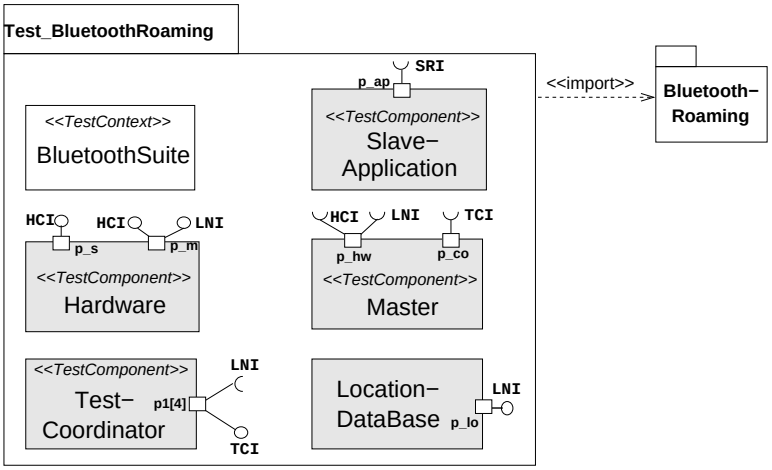


Figure 4.13: Test Package

The test package contains a test context class called BluetoothContext (Figure 4.14). A test context defines attributes and operations needed for test modeling. Test cases are also listed in the text context. A test case is an operation returning a verdict. Additionally, a test log file called BT_LogFile is defined to record test relevant events, e.g. for offline test evaluation. The log file is attached to the test context to indicate that the events of all test cases defined in this test context are recorded in this file.

In addition to the definition of variables and operations, a test context also comprises test configuration and test control. Figure 4.15 shows the Bluetooth test configuration modeled according to the architecture introduced in Section 4.4.1. In this Composite Structure Diagram, both the SUT and the

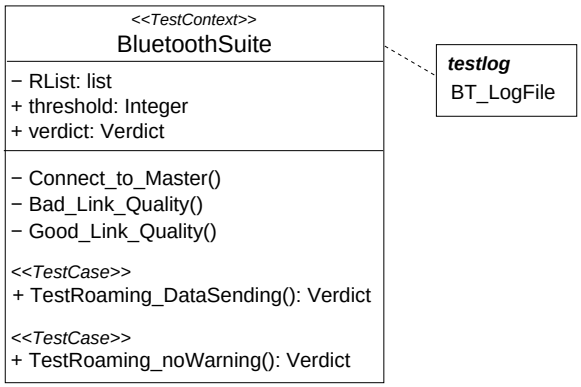


Figure 4.14: Test Context Class

test system are specified. They are connected to each other via ports and connectors[5]. This test configuration on which our test will run corresponds to the architecture depicted in Figure 4.12, but extends the network by four masters.

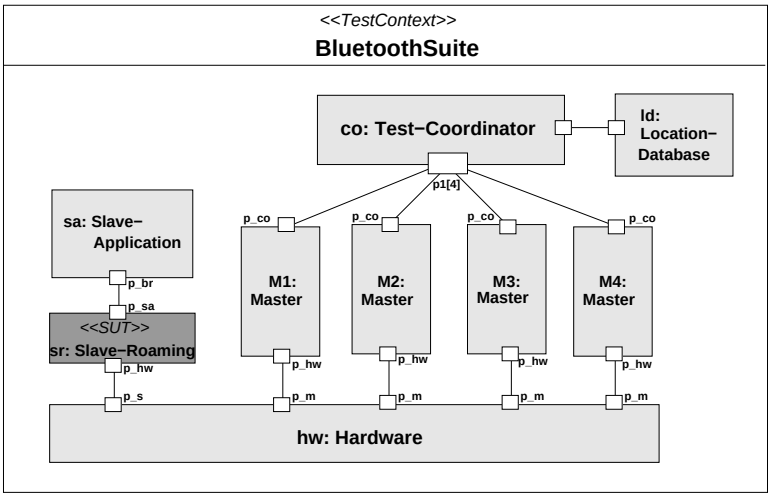


Figure 4.15: Test Configuration Defined in the Test Context Class

The overall test behavior can be controlled by modeling a test control. The Interaction Overall Diagram in Figure 4.16 specifies the test control and indicates the execution order of the roaming test cases. The test starts with the execution of test case TestRoaming_DataSending. If its test result is a

[5]Connectors are the lines between the ports.

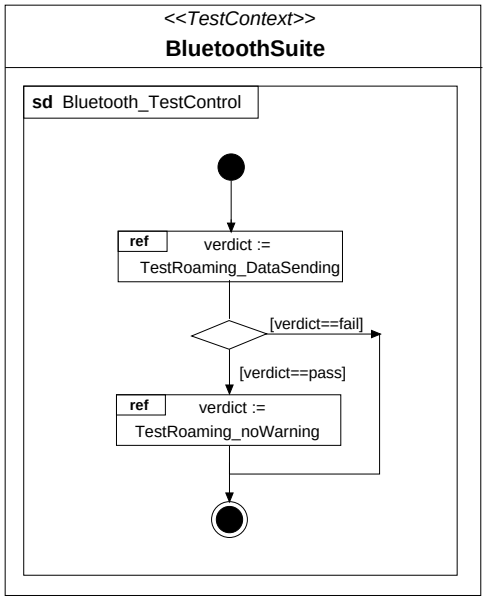


Figure 4.16: Test Control Defined in the Test Context Class

pass, the second test case called TestRoaming_noWarning is carried out. If test case TestRoaming_DataSending fails, the test is finished.

So far, the test architecture including the specification of a test package, a test context consisting a test configuration and a test control have been specified. Next, concrete test behavior will be considered.

4.4.3 Modeling Test Behavior

In Section 4.4.1, the main functionalities of the roaming layer to be tested are listed. In order to test a system extensively, a set of test cases has to be specified in a test model. The following scenario exemplifies one test case. It combines several of the mentioned roaming functionalities to one test case:

> *In the beginning, a connection is established between the Slave and Master M1. After the exchange of two data packages, the link quality between Slave and Master M1 becomes bad. The first alternative master in the roaming list M2 cannot be reached since the link quality is also weak. Thus, after at most two seconds, a further Master M3 is chosen from the roaming list and the connection is established successfully.*

In this scenario, it is tested that the roaming layer of the Slave, first, is able to choose a new master whenever the link connection gets weak, second, requests for a connection establishment to the new master, and last, receives

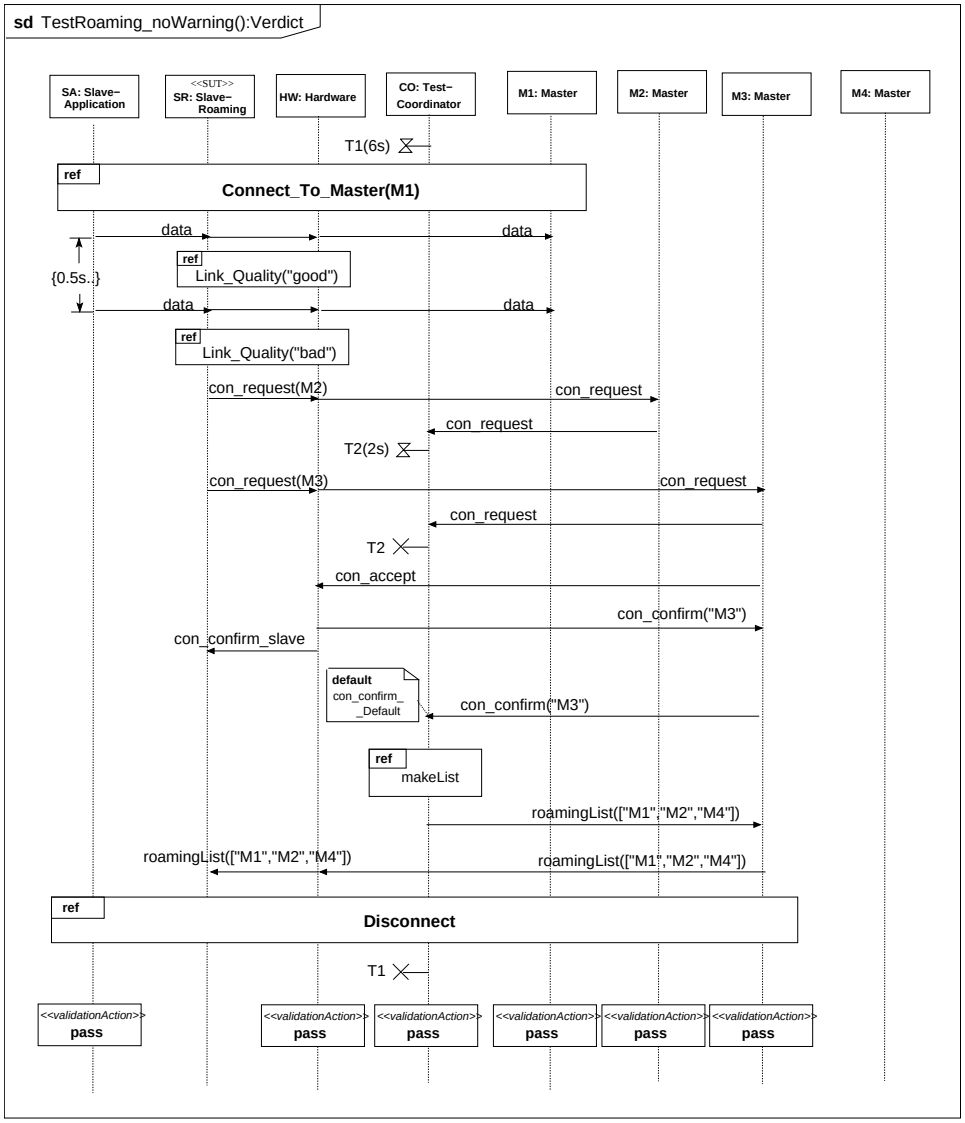


Figure 4.17: Roaming Test Scenario

a connection confirmation from the new master after a successful connection establishment. Furthermore, the connection request is repeated within two seconds if no connection confirmation comes from the new master and the connection to another master is asked. The functionalities of sending a warning to the environment if no new master can be found on the roaming list and the data forwarding feature are not tested in this test scenario. They need to be covered by other test cases in the test model.

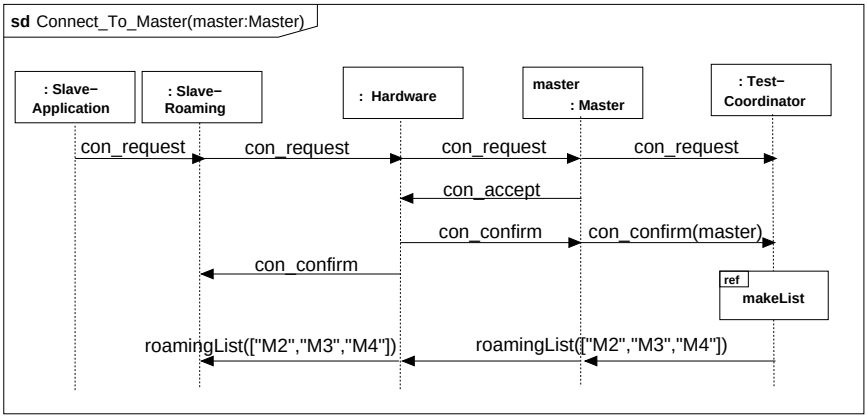


Figure 4.18: Connect to Master Function

The Interaction Overview Diagram illustrated in Figure 4.17 specifies the test case for the scenario described above. The test case is derived from the UML 2.0 behavior diagrams of the system model in Section 3.2. The test case is called TestRoaming_noWarning. Its test behavior starts with the activation of a timer named T1 with the duration of six seconds by the Test-Coordinator instance. T1 is a test component timer started at the beginning and stopped at the end of the test component behavior to assure that the test finishes properly even if e.g. the SUT crashes or does not respond anymore. In case that the execution of this test case takes more than six seconds, a timeout is triggered by Test-Coordinator.

The function Connect_To_Master referenced at the beginning of the test case establishes a connection between the SUT and Master M1 (Figure 4.18). The connection request triggered by the con_request message is initiated by Slave-Application and then forwarded to the master. After the Test-Coordinator is informed about the connection request, the master accepts the connection by sending the con_accept message, resulting in a connection confirmation sent from the Bluetooth hardware to both the slave and the master. Thereupon, the master again informs the Test-Coordinator about the successful connection, which allows the Test-Coordinator to build a new roaming list containing the masters and to transfer it via the master to the slave using the message roamingList(["M2","M3","M4"]) containing the new roaming list. The entries of the roaming list indicate that if the connection between slave and its current master gets weak, Master M2 should be tried next. If this connection cannot be established, Master M3 should be contacted. As a last alternative, M4 should be chosen. If none of the alternative masters can be connected to, warnings would be sent out.

When the referenced behavior of Connect_to_Master is performed according to the specification shown in Figure 4.17, the slave has been successfully

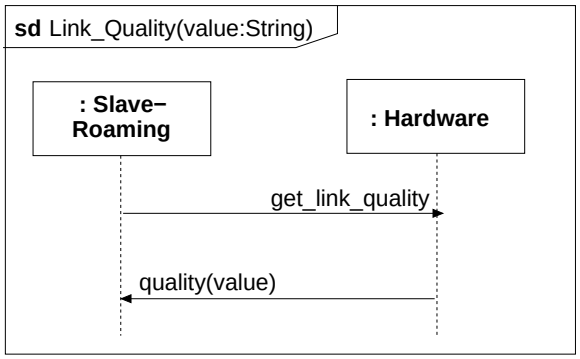


Figure 4.19: Link Quality Evaluation Function

connected to Master M1. Now Slave-Application can send data to Master M1 every 0.5 seconds. Checking the link quality is specified in function Link_Quality. Herein, Slave-Roaming triggers the evaluation request and receives the result from the hardware (Figure 4.19).

In the first evaluation of the link quality, the Hardware is tuned to report a good link quality (Figure 4.17). Thus, further data packages can be sent to Master M1. In the second link evaluation, the quality is determined to be bad. Therefore, a new master is required. According to the roaming list of the Slave, the new master is Master M2. Thus, a connection request is expected to be sent to Master M2 by the SUT. As soon as the connection request is observed and reported to the Test-Coordinator, a timer T2 of two seconds is started to assure that if the SUT cannot establish a connection to the determined master, it chooses a further master and tries to connect to it within two seconds. If the SUT requests a connection to the correct Master M3, the timer T2 is stopped by Test-Coordinator. In the given test case, the connection is accepted (con_accept) by Master M3 and hence confirmed to the SUT and the new master. After the Test-Coordinator noticed the successful connection establishment to the Master M3, it assembles a new roaming list for the slave and sends it via master to the slave. In case that no connection confirmation is received, the default behavior con_confirm_Default is invoked. Finally, slave and master are disconnected, the guarding timer T1 is stopped and the verdict of this test case is set to pass.

Besides the expected test behavior of test case TestRoaming_noWarning, default behavior are specified to catch the unexpected observations which may lead to a fail or a inconclusive test verdict. The given test case uses a default called con_confirm_Default. Figure 4.20 shows how default behavior can be specified both as an Interaction Overview Diagram and as a State Machine[6].

[6]The semantics of both diagrams are identical.

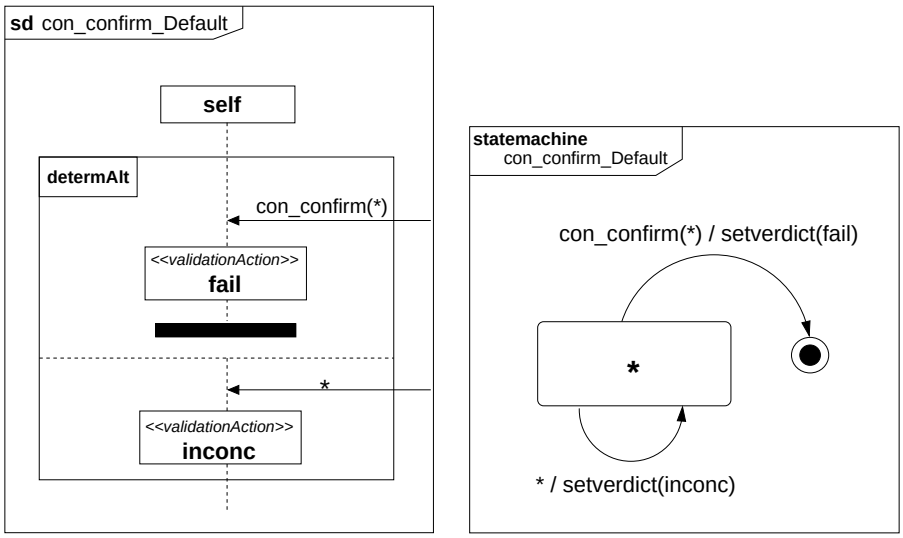


Figure 4.20: Test Defaults

The con_confirm_Default is an message-based default attached to the connection confirmation message in Figure 4.17. In this default behavior, the U2TP-specific operator *determAlt* is applied in order to describe the alternatives[7]. For the Test-Coordinator, this default is invoked if either the connection confirmation (con_confirm) with a wrong parameter or another message than the connection confirmation is received. In the first case, the verdict is set to fail and the test component finishes itself. In the latter case, the verdict is set to inconclusive and the test returns to main test behavior where the default has been invoked.

Test case TestRoaming_noWarning tests functional behavior of the Bluetooth roaming application. The modeling of real-time test requirements is exemplied as follows:

> *In the beginning, a connection is established between the Slave and Master M1. Afterwards, 100 data packages are sent every 0.5 seconds from the application layer.*

This test scenario evaluates that the roaming layer receives data packages from its application layer and forwards these to the hardware layer. Figure 4.21 shows the specification of test case TestRomain_Data_Sending.

In order to specify the evaluation of delay and average jitter within the U2TP model, time is logged both after the application layer has sent the

[7]By modeling with *determAlt*, the order in which the alternatives are considered can be defined.

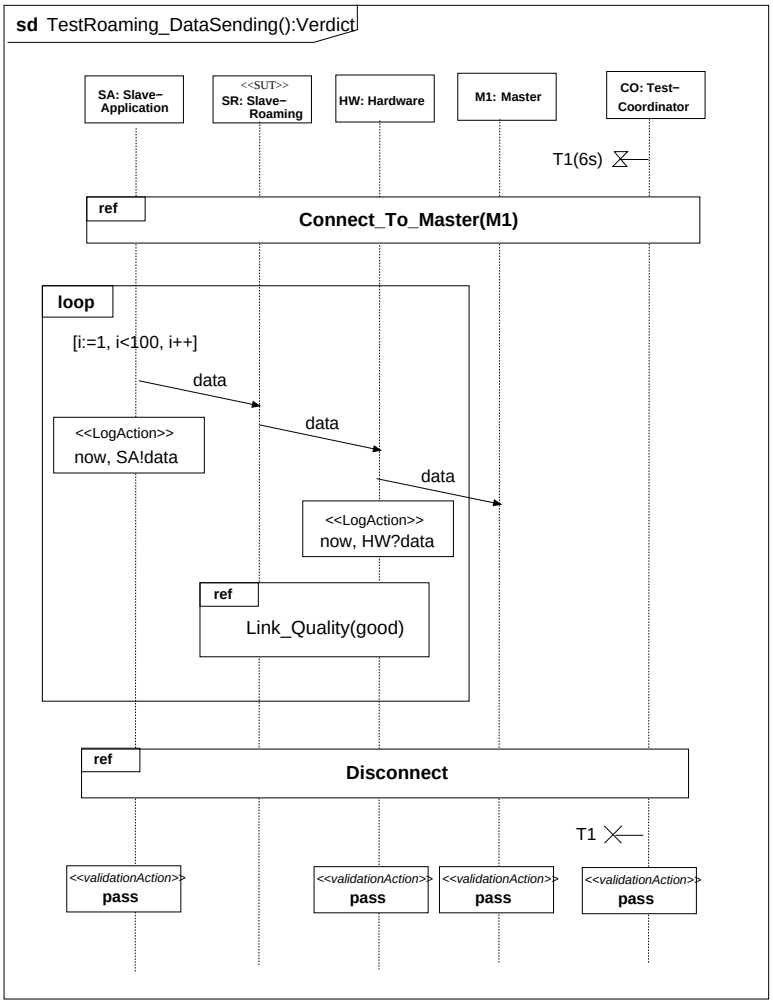


Figure 4.21: Data Forwarding Test Scenario

data packages and after the hardware layer has received the data packages. The logging is modeled by U2TP LogActions[8]. The logged entries are saved in a log file specified for the test context class (Figure 4.14 on Page 72). The format of the logged values is user-defined. In this Interaction Overview Diagram, a logged entry shows the logged time value evaluated by the now expression provided by UML 2.0, the name of the event and the data package number[9]. These logged information is needed for the real-time evaluation specified in the test control.

[8] LogActions are modeled by an action symbol with the stereotype <<LogAction>>.
[9] Integer i is the data package index.

With the logged information of this test case, delay and jitter can be evaluated. While calculating the time difference between the sending and the receiving of the same data package, the delayed time needed by the SUT for data forwarding can be measured. The mathematical formulae[10] for measuring delay is shown below. The function implementing the formulae can be called by the U2TP arbiter.

The following formulae shows the delay of the i-th data package, whereas $t_{HW?Data}(i)$ is the time point where the i-th data package is received by the hardware layer and $t_{SA!Data}(i)$ defines the time point where the data package is sent by the appliation layer:

$$Delay\_DT(i) := t_{HW?Data}(i) - t_{SA!Data}(i)$$

The average jitter provides the average variation of periodical occurrences. As a precondition, the sending of data packages must be performed equidistantly. According to the specification, data packages are sent every 0.5 seconds, i.e. $\forall i : |t_{HW?Data}(i+1) - t_{HW?Data}(i)| = 0,5s$. The mathematical formulae for average jitter is:

$$Jitter\_JT := \sum_{i=2}^{n} |(delta - (t_{HW?Data}(i) - t_{HW?Data}(i-1)))|/(n-1)$$

$$with\ delta := \sum_{i=2}^{n} (t_{HW?Data}(i) - t_{HW?Data}(i-1))/(n-1)$$

4.5 Concluding Remarks

In this chapter, the U2TP concepts have been introduced and explained in detail. U2TP extends and augments UML 2.0 for test model specification. This profile bridges the gap between designers and testers by providing a means to use UML 2.0 for both system and test specification. Moreover, U2TP allows the reuse of UML 2.0 system design documents for testing and integrates test development in an early software development phase.

The specification of a U2TP test model has been examplified for the Bluetooth roaming case study. The case study shows that U2TP is suitable for functional and real-time test modeling. Based on the U2TP test model, executable test code can then be generated.

[10] The "!" character symbolizes the sending of a message and the "?" the reception of a message.

Chapter 5

Transforming System Model to Test Model

"Everything should be made as simple as possible, but not simpler."
– Albert Einstein

This chapter provides a guideline to use U2TP concepts by retrieving a U2TP test model from an existing UML 2.0 system model. The guideline steps are realized by concrete transformation rules written in the transformation language called *Tefkat*. The applicability of the transformation rules is demonstrated on the Bluetooth roaming case study.

5.1 Related Work on UML-Based Test Generation

Model-based testing is not a new research area in the testing domain. In [BJK+05], a collection of research works done in this area can be found. Most of these works are set upon systems specified in Finite State Machine (FSM), Labelled Transition System (LTS), Specification and Description Language (SDL) or Message Sequence Chart (MSC) etc. [LY96, Sif89, TPvB97, ITU99, Z1299, GKSH99a, GKSH99b, GH02, SEG+98, BBJ+02]. According to the classical test methodology, two different aspects have to be distinguished: the modeling of SUT and the definition of test system features. For testing based on UML system models, different approaches and tools are discussed in the following.

In [Bin99], Binder describes test requirements derived from UML models. The derivation is supported by test design patterns which focus on determining appropriate test strategies, faults that may be detected, and test case development from model and oracle. The test is developed for different scopes in the implementation.

Briand and Labiche introduce a system test methodology called Testing Object-orienTed systEms with the Unified Modelling Language (TOTEM)

where test requirements are derived from UML analysis artifacts such as Use Cases, their corresponding Sequence and Collaboration Diagrams, Class Diagrams and the use of the Object Constraint Language (OCL) [BL01]. Subsequently, the test requirements are transformed into test artifacts by using more detailed design information.

Offutt and Abdurazik propose two approaches for software testing using a restricted form of State Machines and Collaboration Diagrams of a UML system model [OA99, AO00]. The goals of the approaches are to test the system models themselves and to use information from different types of diagrams during testing.

Scheetz et al. depict an approach to generate system test from UML Class Diagrams [SvMF+99]. For that, a restricted form of UML Class Diagrams is used to represent SUT architecture. From the Class Diagrams, test objectives are derived. However, when multi-object test objectives are aggregated, the space of possible test objectives becomes very complex. By means of test criteria, the complexity can be reduced [ARFC03].

Born et al. derive test systems for component-based SUTs [BSKH04, BSL00]. Test components are developed as soon as system component interfaces and the overall system architecture are defined. A Test Framework (TFW) contains base types of test components and generic component behavior define test configuration, test initiation, coordination exchanges and test results.

A number of tool are involved in the automatic test generation process. Each performing its individual task during manual or (semi-)automatic transformations. In the AGEDIS project, a UML-based test tool chain is developed to reuse and adopt existing system validation and test code generation tools [Cav01, HN04, HN03].

The approaches mentioned above mainly rely on UML 1.x. Moreover, they only allow certain UML diagram types for test code generation, e.g. Collaboration Diagram and Use Cases. Some of them even restrict or refine the concept space of the UML diagram types. Also, the gap between the system development and the test development exist, since the developers still use a different language for system modeling than for test modeling. In this thesis, we shall generate U2TP test models utilizing UML 2.0 for system modeling without constraining the expressiveness of UML 2.0. A new technique is provided in this work, where test-specific requirements are adjoined when retrieving a test model from its system model.

5.2 A Guideline for Test Modeling with U2TP

In the previous chapter, the test modeling language U2TP has been introduced. However, the U2TP standard specifies a collection of test modeling concepts without providing a guideline for utilizing these concepts or

a methodology of how to specify test models properly. This section tries to close this gap by presenting a guideline for specifying test models with U2TP. When doing so, the guideline aims at modeling tests in an efficient and effective way. To achieve this, test models are derived from existing UML 2.0 system models. In this way, a single model can be utilized for both system and test developments [Dai04b].

In order to describe the derivation steps effectively, the U2TP concepts are subdivided into four categories, as shown in Table 5.1[1]: The *Mandatories* are obligatory concepts which identify a U2TP test model. *Optionals* can further be distinguished between normal and derivable concepts. An example for derivable mandatory concepts is a test case which can be deduced from system scenarios. Likewise, test configuration can be derived from system configuration. But since a test configuration is not essential for a test model, it is classified as a derivable optional concept. The concrete techniques to derive test diagrams from system diagrams are explained later in this chapter[2].

Test Architecture Concepts	Test Behavior Concepts	Test Data Concepts	Time Concepts
SUT	Test objective	Data pool	Timer
Test component	Test case	Data partition	Time zone
Test context	Defaults	Data selector	
Test configuration	Validation action	Wildcards	
Test control	Verdicts	Coding rules	
Arbiter			
Scheduler			

Mandatories | Derivable Mandatories | Optionals | Derivable Optionals

Table 5.1: A Guideline Schema on U2TP

In the following, the mandatory and optional concepts and their possible derivations are outlined. A U2TP test model may use all UML 2.0 diagram types for test specification. Depending on the given diagram types in the system model, test model diagrams are obtained.

[1]The four categories are presented in different textures in the figure.

[2]According to the U2TP standard, the scheduler is to be implemented by the tool vendor and is therefore not classified for this guideline.

5.2.1 Prerequisites for System Models

Before defining the guideline, further thoughts are investigated regarding prerequisites for the system models. The assumptions which UML 2.0 system model has to fulfill to enable an applicable transformation process later on are as follows:

- A system model provides the information obligatory for the transformation of the test model skeleton. However, the relevance of the resulting test model depends on the preciseness of the initial system model. In other words, the more specific a system model is, the more concrete is the resulting test model skeleton[3].
- The model transformation addresses systems where both SUT and test system can be retrieved from an existing system model. To do so, the system model must be self-contained and should not communicate with its environment.
- In order to be able to trigger and observe the SUT, communication connections must exist between the system components which should become SUT and test components. Moreover, the interface between the SUT and its test component must provide at least one public method with return value, so that the SUT can be triggered by the method call and observed by its return value.

5.2.2 Test Preparation

Before starting with the test model derivation from system model, some preparational work must be done. Herein, it is important:

- to define a new UML 2.0 package as test package, and
- to import the system package in order to get access to its classes and interfaces.

5.2.3 Deriving Test Architecture

I. Mandatories:

- Create a new class in a Class Diagram and assign it to *SUT* class by stereotype <<SUT>>. If multiple system classes are to be tested, they become nested classes defined in the SUT class.

[3] In the following, when talking about test model, the test model skeleton is meant.

- Create a new instance in an Object Diagram referencing to the SUT class as type. Assign the instance to *SUT* instance by stereotype <<SUT>>.
- Specify a *test context* class in a Class Diagram stereotyping with <<TestContext>> and list all the test-relevant attributes and test cases. If test control and test configuration exist, they are also defined in the test context.

II. Optionals:

- When performing component or integration test, a test system is required to stimulate and observe the SUT. Herefore, create a new class in the Class Diagram and assign it to *test component* class by stereotyping <<TestComponent>>. If multiple system classes should become one test component class, the system classes become nested classes in the test component class. Beware that the system classes which become the SUT class are not chosen as a test component class.
- Create a new instance in an Object Diagram and assign it to *test component* instance. The type of the instance must be a test component class.
- Having a Use Case Diagram, the concrete behavior of the use cases are represented by supplementary behavior diagrams. Thus, each use case specifies a test scenario. For the specification of *test control*, reference to the use cases as test cases and connect them in an Interaction Overview Diagram.
- Interaction Overview Diagrams provide good means to define *test control*. To do so, identify existing test cases in the test model. Create an Interaction Overview Diagram in the test model and create references for the test cases. Connect the references by control flow in the Interaction Overview Diagram. In a more complex test control specification, loops and conditions should also be used.
- *Test configuration* is best specified in a Composite Structure Diagram. If system model already specifies its configuration by a Composite Structure Diagram, then adopt this diagram and align it according to the test configuration defined in the test requirement.
- *Test configuration* can also be retrieved from Class Diagrams. Herein, the interfaces between the relevant test components and SUT are connected with each other according to the test requirement.

- *Test configuration* can also be built from system scenarios illustrated in Interaction Diagrams. To do so, whenever two components exchange messages with each other, create a communication channel between the components in the test configuration diagram.

5.2.4 Deriving Test Behavior

I. Mandatories:

- For the specification of *test cases*, adopt given Interaction Diagrams from the system model. Stereotype the diagrams with <<TestCase>>. Rename or group lifelines in the diagram according to test configuration. Assign them by their roles (i.e. test component or SUT). When lifelines are grouped in a test case, internal messages can be neglected. But the external message exchanges of the grouped lifeline to its neighbouring lifelines must be adopted to the test case specification.
- *Test case* can also be derived from system behavior defined in State Machines. Since State Machines specify the complete local behavior of a component, simulation is needed to retrieve communication traces between the component. Each simulated trace is stored in an Interaction Diagram and represents one test case.
- *Verdicts* can be derived from system scenarios where valid system behavior are defined. Verdicts are set for test components at the end of a test case specification. In an Interaction, it is defined at the end of a lifeline. *Verdicts* can also be defined in a State Machine. There, it is attached to transitions. Usually, a verdict within a test case is set to pass. The setting of other verdict values (i.e. inconc, fail, error) are specified in default behavior.

II. Optionals:

- Write a *test objective* for each test case which should be specified. According to the U2TP standard, test objectives do not have to be illustrated by diagrams and can also be in prose.
- Unexpected system behavior during the test run is caught by *defaults*. State Machines are good means to define defaults in order to cover the overall behavior of one component. Use *wildcards* to catch unexpected events. Verdict settings defined in a default behavior are either fail or inconclusive.

- Interaction Diagrams can also be used for defining *defaults*. Herefore, alternatives are defined distinguishing between behavior which should be tested leading to a failure or inconclusive behavior.
- After having specified default behavior in diverse diagrams, they must be activated for the appropriate suppliers. To do so, attach the defaults to their suppliers.
- In case that a test system owns a user-defined arbiter[4], *validation actions* are required in the test case specification in order to exchange verdicts between test components and the arbiter. They are added in the test cases by stereotyping the verdict setting with <<validationAction>>.

5.2.5 Deriving Test Data

I. Optionals:

- A *data pool* consists of all possible test data and their type definitions. They can be copied directly from the system model or be imported from a data package of the system model. The specification of the data pool is best illustrated in a Class Diagram or an Object Diagram. Concrete test data can also be derived from system scenarios by collecting all data exchanged between the system components and put them in the data pool class. Stereotype the data pool class with <<DataPool>>.
- A *data partition* can be derived from a group of system scenarios where the equivalence classes correspond to the test scenarios. For the specification of data partitions, Class Diagram are recommended.
- *Wildcards* are used in defaults for catching unexpected system responses and thus utilized in the default diagrams.
- The definition of *data selectors* correspond to the specification of the test requirements. Class Diagrams are recommended for data selector specification.
- Provide *coding rule* information according to test requirement. This can be done either within a Class Diagram or a Composite Structure Diagram.

[4]The U2TP standard defines a mandatory default arbiter with its arbitration algorithm to be provided by the tool vendor. In this guideline, we concentrate on the definition of an user-defined arbiter which is optional.

5.2.6 Deriving Time

I. Optionals:

- Generally, for each trigger and response between a test component and the SUT, *timers* can be started before and stopped after the response. For that, both Interaction Diagrams and State Machines can be used. The timer duration must be provided by the tester. Assure that the timer is declared as an attribute within the test component class whenever a timer is specified in a test case. Beware that a timer is not defined for the SUT since its behavior is not changeable.
- *Timers* can also be derived from time constraint specification of a system scenario. Herefore, the timer replaces the time constraint. A timer is started where the time constraint begins and is stopped at the end of the time constraint. The expiration duration of the timer should be of the same value like the time constraint.
- If the test system is a distributed system, assign *timezones* to the test components defined in the test configuration diagram.

5.3 Transforming System Models to Test Models

Following the guideline steps introduced in the previous section, a tester may reuse an existing system model and derive a test model from the system model manually. A more ambitious effort than the manual test modeling is to generate the test model automatically. In the following, we introduce an approach to automatic test model generation realized by a model transformation.

Figure 5.1 shows the implementation architecture for the U2TP model transformation supplied in this work[5]. There, three metamodels based on MOF 2.0 exist. They are called the *UML 2.0* metamodel, the *Test Directives* metamodel and the *U2TP* metamodel. The *Test Directives* metamodel enables the integration of test requirements during transformation. It is explained in Section 5.3.1 in detail. Both UML 2.0 and the Test Directives metamodels are source metamodels for the definition of the transformation rules whereas the U2TP metamodel serves as the target metamodel. Consequently, model transformation is carried out on UML 2.0 and Test Directives model instances to get U2TP model instances [Dai04a]. The test model transformation rules are realized in *Tefkat* (Section 2.1.3).

[5]Transformation ruls are defined on metamodel level whereas the model transformation is performed on model level.

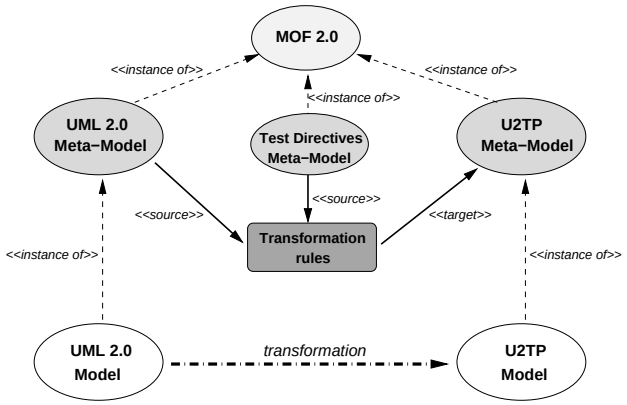


Figure 5.1: Implementation Architecture for U2TP Model Transformation

The test model transformation from system model by means of transformation rules only generates a skeleton of a test model. Test-specific requirements cannot be retrieved from the system model. In order to accomplish the skeleton to a feasible test model, further modifications and refinements on the test model skeleton must be made. In the following, we give an insight into the realization of the test model transformation.

5.3.1 Integrating Test Requirements During Transformation

The starting point of our test model transformation is an existing UML 2.0 system model. However, test-specific requirements directing the test model specification, such as the choice of SUT component, configuration of the test system, selection of test scenarios, specification of timers, invocation of default behavior, setting of test verdicts, definition of timezones, coordination of test components etc., cannot be derived from the system model. They must be defined explicitely and included in transformation.

In order to integrate test requirements during test model transformation, a MOF 2.0-based metamodel called *Test Directives* is developed in this work. The term *Test Directives* denotes a collection of test-specific information which, when combined with the system model, derives a test model. Thus, a Test Directives model provides a formal specification of test requirements which is supplied to the transformation rules.

Figure 5.2 shows the application areas of both Test Directives model and U2TP model and their allocation within the test development process. While a Test Directives model mainly supplies the requirements of a test model, U2TP provides good means to design and specify the tests. Nevertheless, a Test Directives model also influences the test design when e.g. assigning

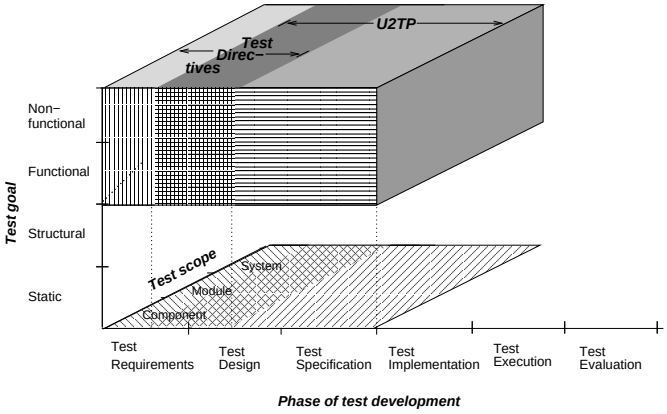


Figure 5.2: Application Area of Test Directives and U2TP

roles of components for test configuration. Also, depending on the model abstraction level, U2TP can be used for test requirement specifications. Thus, their applications depicted in Figure 5.2 are overlapping. In the following, the definition of the Test Directives metamodel is illustrated. Their relevance and application are shown later in the definition of the transformation rules in Section 5.3.2.

The Test Directives metamodel provides means such as to assign roles to components, to group classes, instances or lifelines, to rename components, to select and specify test scenarios, to choose and activate default behavior, to add events in test behavior, to define timers, and to add components to the test system etc. Figure 5.3 shows the package dependencies[6] between the Test Directives, the UML[7] and the Infrastructure Library metamodels[8]. The top-most element in a Test Directives model is a TestDirectiveSet[9] (Figure 5.4). It has a name and consists of TestDirectiveElements. Meta-class TestDirectiveElements is a refinement of the MOF 2.0 metaclass ENamedElements and thus are also named. Furthermore, it refines other Test Directives meta-classes.

The meta-class Grouping is needed to merge classes, instances and lifelines in a test model diagram (Figure 5.5). It has an attribute called groupingPurpose to indicate the role of the new entity. Role is an enumeration of SUT, TestComponent and Arbiter. Meta-class Grouping refines three meta-classes called GroupingClasses, GroupingInstances and GroupingLifelnes. Figure 5.5

[6]Dependencies are illustrated by a dashed arrow.

[7]The UML package defines the UML 2.0 metamodel.

[8]By referencing to the latter, the Test Directives metamodel becomes a MOF 2.0 based metamodel.

[9]The complete Test Directives metamodel can be found in Appendix A.

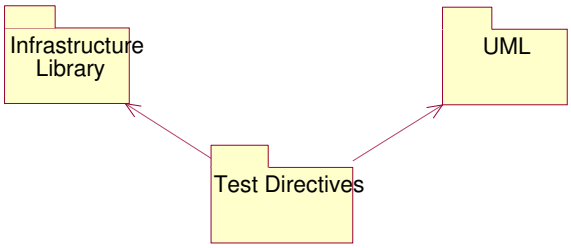


Figure 5.3: Package Dependencies of the Test Directives Metamodel

also shows the specification of meta-class GroupingClasses. With this Test Directives meta-class, at least two classes in the system model can be merged, indicated by the cardinality of 2..*. Whenever a new class in the test model is defined, a new name is assigned by the attribute newTestClassName.

By means of the Grouping meta-class, the resulting test class is renamed accordingly to the Test Directives specification. However, there might also be adopted classes in the test model without performing grouping with other classes. Still, they need to be renamed and assigned with roles. To do so, meta-class Renaming shown in Figure 5.6 is defined. It provides the new name and new role to classes, instances and lifelines.

The Test Directives metamodel also provides means to add supplementary

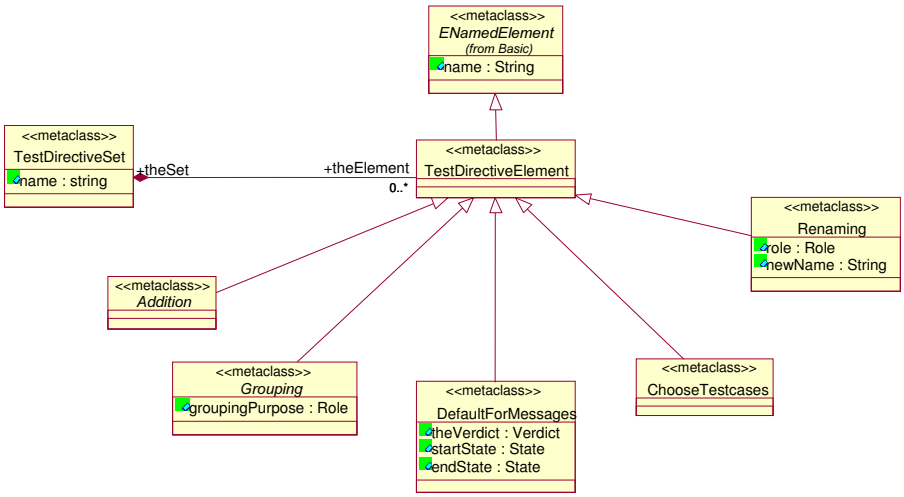


Figure 5.4: Overview on Test Directives Metamodel Classes

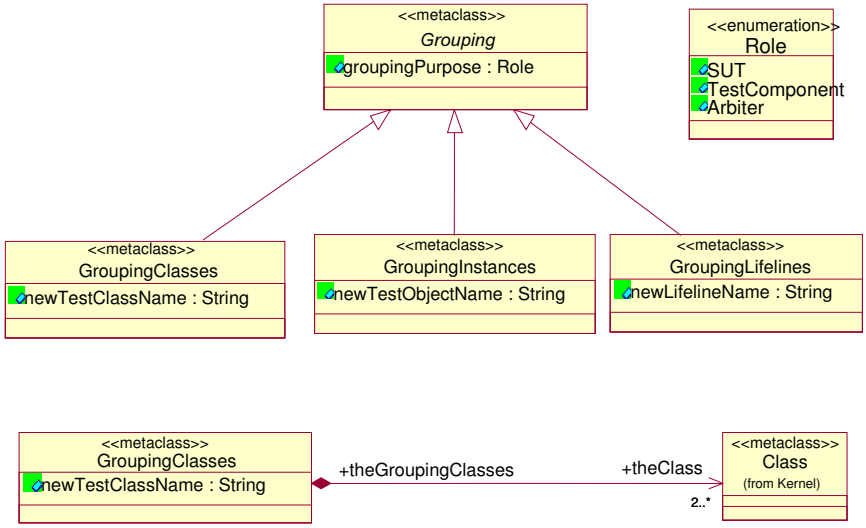


Figure 5.5: Grouping System Model Classes

elements to the test model (Figure 5.7). One means is to add components, e.g. an arbiter, to the test system. Figure 5.8 shows the definition of a AddNewComponent meta-class which is able to create a new component within the test configuration and connect the new component to its adjacent components via appropriate ports.

If a new component is added to the test configuration, the behavior of the newly created component must be added and specified. Meta-class AddNewEvent enables adding of new events to existing test scenarios (Figure 5.9).

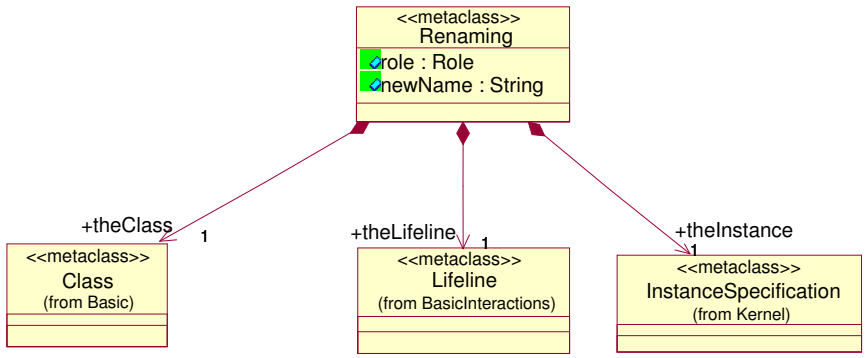


Figure 5.6: Renaming Classes, Instances and Lifelines

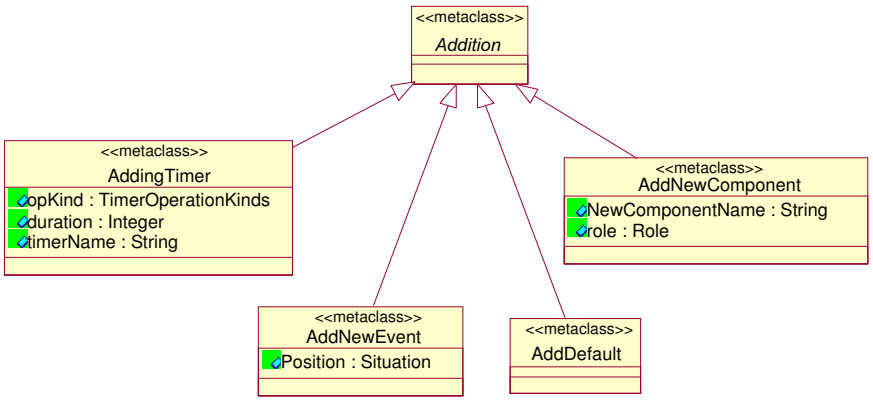


Figure 5.7: Adding New Elements

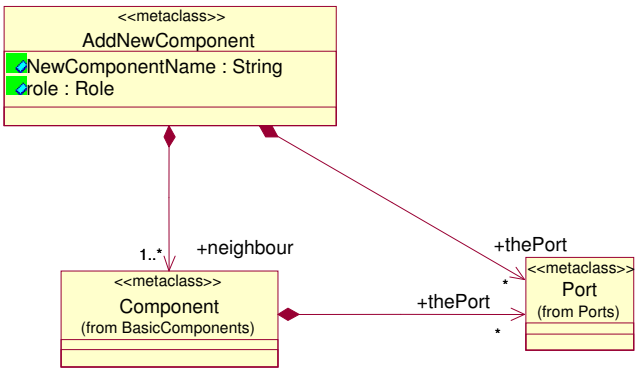


Figure 5.8: Adding New Components to the Test Architecture

The location of the new event is associated with an existing event and lifeline in the test scenario. Herefore, it is also important to determine whether the new event should be positioned before or after the existing event to avoid ambiguity.

In order to derive test cases from system scenarios, the appropriate behavior needs to be selected[10]. After a test behavior is chosen, the whole scenario must be modified according to test requirements specifications.

When refining a test scenario by timers, it is necessary to determine when the timer starts and stops. Figure 5.10 shows the definition of meta-class AddingTimer. Herefore, the location of the timer is associated by a lifeline and a message. A timer can either be attached before or after a message.

[10]This is done by the Test Directives meta-class ChooseTestcases associated with an existing behavior diagram, to be found in the Appendix.

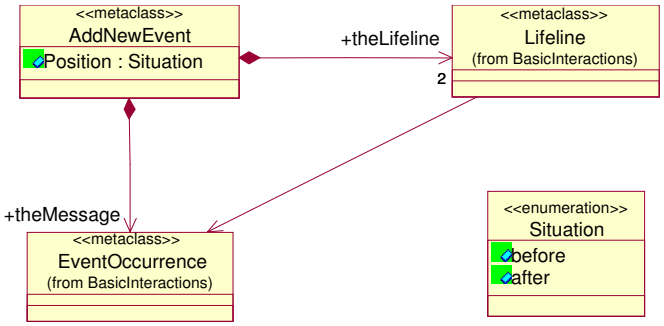


Figure 5.9: Adding New Events to Test Scenario

For simplicity, we agreed that a timer is always attached *after* the given message. Operations to start or stop a timer is given by attribute opKind and its duration is specified by attribute duration.

Default behavior specification is associated with State Machines. When activating default for a certain message defined in the test behavior, metaclass DefaultForMessages is needed (Figure 5.11). The Test Directives model provides the information about the name of the message for which the default behavior should be specified. The transition in the default specification is indicated by the startState and endState, with the dedicated message as its transition trigger. At last, the appropriate default behavior is activated and attached to the test case.

Below, an example for the collaboration of UML 2.0 model and Test Directives model during model transformation is shown. Assuming there is

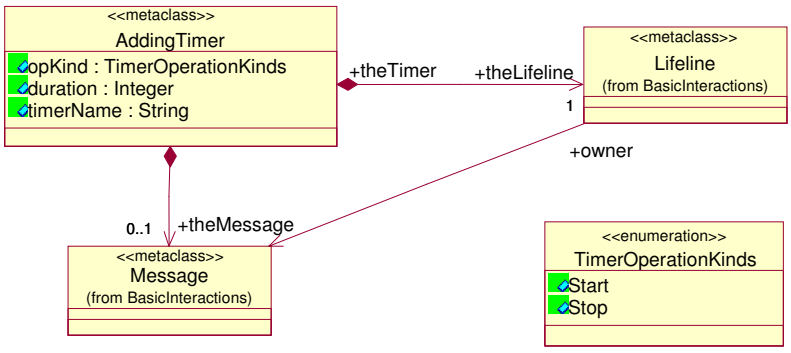


Figure 5.10: Adding Timers to the Test Scenarios

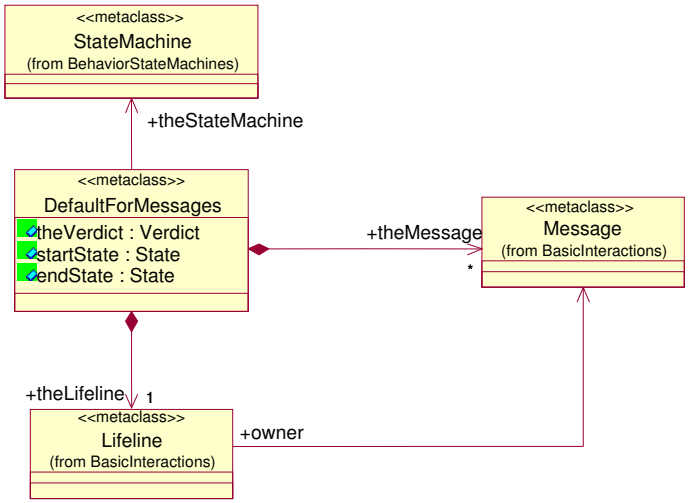


Figure 5.11: Specifying Default Behavior for Messages

an existing Class Diagram in the system model, U2TP concepts should be applied. According to the methodology introduced in Section 5.2, one of the test architecture mandatories says:

> *" . . . create a new class in the Class Diagram and assign it to* test component *class by stereotyping <<TestComponent>>. If multiple system classes should become one test component class, the system classes become nested classes in the test component class . . . "*

Thus, besides the Class Diagram, the grouping mechanism provided by the Test Directives metamodel is needed. Figure 5.12 shows how the transformation can be performed in this case. On the left upper corner, a UML 2.0 package UMLPkg with three classes is shown. In the left lower corner, the Test Directives package TestDirectivePkg is shown, specifying the grouping of Class1 and Class3 to a test component class. For the graphical representation of the Test Directives model elements, we oriented to existing UML 2.0 symbols. Thus, after the transformation, the output test model U2TPPkg on the right side of the figure specifies a test component named myTC, with Class1 and Class3 classes being nested. The application of other Test Directives elements is demonstrated in the following subsections more concretely when explaining the model transformation rules.

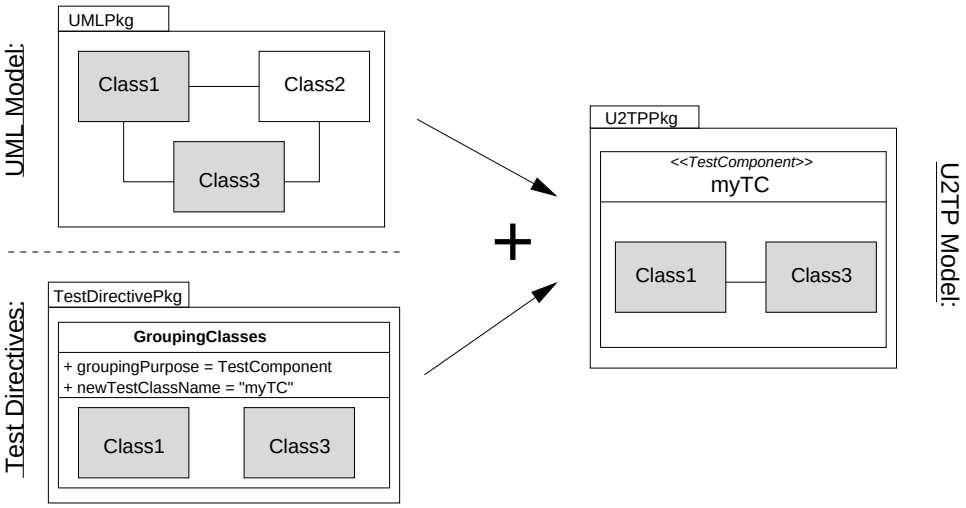


Figure 5.12: Creating a Test Component Class

5.3.2 Transformation Rules for the Test Model

This section provides the concrete definition of the U2TP model transformation rules realized by the Tefkat transformation language. In the following, the transformation rules are described in a figure consisting of two parts: a header part and code part. In the header part (upper part of the schema), the aim of the transformation is given. In the code part (lower part of the schema), Tefkat code is shown. These complete transformation rule catalogue can be found in Appendix B. By means of the Bluetooth Roaming case study, the application of the rules are exemplified.

5.3.2.1 Transformation Preparation

Starting with the definition of the transformation rules, Figure 5.13 shows a model transformation file called UMLToU2TPTransformation. The transformation takes two source models called UMLsrc and TestDirSrc in order to transform it to the target model U2TPtgt (Lines 1–2). Their metamodels needed for the definition of the transformation rules are imported in Lines 4–8.

Creating Test Model and Applying U2TP Profile At the beginning of the transformation, a new U2TP model is created. Afterwards, the U2TP profile needs to be applied to the model. A profile must be applied to a model in order to be able to use the concepts and stereotypes defined in the profile. This is done by the Rule ApplyProfile in Figure 5.14. First, the

Description: Transformation definition and its imports.

```
1 TRANSFORMATION
2   UMLToU2TPTransformation: UMLsrc, TestDirSrc -> U2TPtgt
3 IMPORT
4   platform:/resource/My_TestTefkat/src/model/UML2.ecore
5 IMPORT
6   platform:/resource/My_TestTefkat/src/testdirectives_MM/testdirectives.ecore
7 IMPORT
8   platform:/resource/My_TestTefkat/src/model/u2tp.ecore
```

Figure 5.13: Transformation Definitions and Imports with Tefkat

U2TP package is named like the UML 2.0 package, with the prefix "Test_" to indicate that it is a test model (Line 8). The U2TP profile is applied by setting the imported package with the URI of the profile (Lines 9–10). Additionally, a tracking relationship U2TPModel2UMLModel is defined in Lines 1–4 which links the UML 2.0 model with the U2TP model (Lines 11–13). By means of tracking relationships, other transformation rules may reference them for further modification.

Description: Create a test package for the U2TP model. Apply the U2TP profile to the newly created test model. Give the model the name with the prefix "Test_", reference to the U2TP Profile in order to apply U2TP to the UML 2.0 model.

```
1  CLASS U2TPModel2UMLModel {
2      :: uml2::Model theUMLModel;
3      :: uml2::Model theU2TPModel;
4  };

5  RULE ApplyProfile(umlPackage, u2tpPackage)
6      FORALL ::uml2::Package umlPackage
7      MAKE ::uml2::Package u2tpPackage // Create U2TP package
8      SET u2tpPackage.name = append("Test_", umlPackage.name),
9          u2tpPackage.appliedProfile.setImportedPackage =
10     <platform:/resource/My_TestTefkat/src/model/U2TP_Profile.profile.uml2>
11     LINKING U2TPModel2UMLModel
12         WITH theUMLModel = umlPackage,
13              theU2TPModel = u2tpPackage
14         // Linking source model with target model
15 ;
```

Figure 5.14: Apply U2TP Profile to the Test Model

Description: Import UML 2.0 model to U2TP model in order to get access to the class for testing purposes.

```
1 RULE ImportSystemDesignPackage()
2     WHERE U2TPModel2UMLModel
3         LINKS theUMLModel = systemPackage,
4               theU2TPModel = testPackage
5     SET testPackage.packageImport.importedPackage = systemPackage,
6         testPackage.name = append("Test", systemPackage.name)
7                   // Name test package
8 ;
```

Figure 5.15: Import UML 2.0 Model Package to U2TP Model

Importing System Package A UML 2.0 package must be imported by the U2TP model to get access to the system model elements while test execution. Rule ImportSystemDesignPackage in Figure 5.15 references to the tracking relationship U2TPModel2UMLModel and retrieves the source and the target models. The system package is then imported and the test package is named by the prefix "Test_".

Figure 5.16 shows the impact of the transformation rules applied to the Bluetooth roaming case study. It shows the Bluetooth test package after having applied the transformation rules ImportSystemDesignPackage and ApplyProfile. There, the test package called Test_BluetoothRoaming imports the system package BluetoothRoaming and the U2TP profile is applied to the test package.

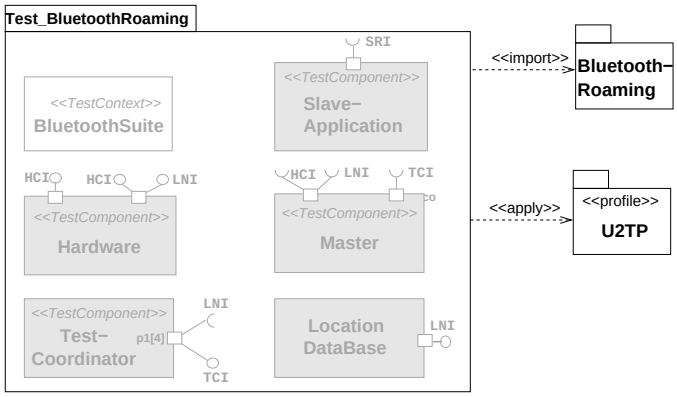


Figure 5.16: Test Package Creation for Bluetooth Roaming

Adopting System Model Elements After having created the U2TP model Relevant UML 2.0 system model elements adopted to the U2TP test model. Rule CopyUMLElementToU2TP shown in Figure 5.17 is an abstract rule which copies UML 2.0 classifiers. For each UML 2.0 classifier, a U2TP classifier is created (Lines 2 and 9). Moreover, all attributes, operations and ports (Lines 3–7) of this classifier are copied to the U2TP classifier (Lines 10–14)[11]. Additionally, the new classifier in the test model is named with a prefix "Test" extending the existing system model classifier (Line 16).

Description: Copy UML 2.0 system model elements to the U2TP test model. Name the elements in the U2TP model with the prefix "Test".

```
1  ABSTRACT RULE CopyUMLElementToU2TP(umlClassifier, u2tpClassifier)
2      FORALL ::uml2::Classifier umlClassifier
3      WHERE umlClassifier.ownedAttribute = attributes
4      // Get all attributes
5      AND umlClassifier.ownedOperation = operations
6      // Get all operations
7      AND umlClassifier.ownedPort = ports
8      // Get all ports
9      MAKE ::u2tp::Classifier@U2TPtgt u2tpClassifier
10     SET u2tpClassifier.$attributes = umlClassifier.$attributes,
11         // Copy all attributes to the U2TP classifier
12         u2tpClassifier.$operations = umlClassifier.$operations,
13         // Copy all the operations to the U2TP classifier
14         u2tpClassifier.$ports = umlClassifier.$ports,
15         // Copy all the ports to the U2TP classifier
16         u2tpClassifier.name = append("Test", umlClassifier.name)
17         // Name the new test model elements
18 ;
```

Figure 5.17: Copying UML 2.0 System Model Elements to U2TP Test Model

Usually, a transformation rule is run by the Tefkat engine automatically. However, an abstract rule is not executed by the engine by default. An abstract rule only affects the behavior of a transformation if it is extended by a non-abstract rule. However, since CopyUMLElementToU2TP is an abstract rule, it is extended by additional non-abstract transformation rules to copy UML 2.0 elements such as a UML 2.0 class. Rule CopyUMLInstancesToU2TP in Figure 5.18 is a concrete rule extending CopyUMLElementToU2TP (Line 2). This rule takes additional features of an instance defined in the system model and targets them to the test model. Besides attributes, operations and ports, it also adopt the instance classifiers, values and specifications in an Object Diagram (Lines 8–13).

[11]Herefore, the rule makes use of the reflection definition mechanism of the Tefkat language in Lines 3–7 using the $ symbol to indicate that what follows is not the literal name of an attribute, but is instead an expression which should be evaluated to yield either the name of an attribute or an instance.

Description: Copying instance specifications to the newly created U2TP model.

```
1  RULE CopyUMLInstancesToU2TP(umlInstance, u2tpInstance)
2      EXTENDS CopyUMLElementToU2TP(umlInstance, u2tpInstance)
3      FORALL ::uml2::InstanceSpecification umlInstance
4      WHERE umlClass.classifier = superclass // Get classifiers
5            umlClass.slot = slot  // Get classifier slots for instance values
6            umlClass. specification = specification  // Get classifier specifications
7      MAKE ::u2tp::InstanceSpecification@U2TPtgt u2tpInstance
8      SET u2tpInstance.$classifier = umlInstance. $classifier
9          // Copy instance classifiers .
10         u2tpInstance.$slot = umlInstance.$slot
11         // Copy instance slots.
12         u2tpInstance. $specification = umlInstance.$specification
13         // Copy instance specification .
14 ;
```

Figure 5.18: Copying System Model Instances to Test Model

5.3.2.2 Transformation Rules for Test Architecture

When a U2TP model is created, transformation rules concerning concrete test architecture need to be defined. This includes e.g. the role assignment of the components, the specification of the test context, the definition of the test configuration, the creation of the test control etc.

SUT and Test Components One of the prerequisites listed in Section 5.2.1 is that the system model is a self-contained and a closed system. Having such a system, both the SUT and the test system are derived from the system model. To do so, the components must be assigned by the roles of either SUT or test component. Figure 5.19 defines a rule to assign system classes with roles. For that, a Test Directives model is needed since information about roles is not included in the system model (Line 2). According to the Test Directives specification, Rule RoleAssignment_Classes creates a new class and assigns it either the U2TP stereotype <<TestComponent>> or <<SUT>> (Lines 3 and 7). The Test Directives model provides the new name of the class as well (Line 12).

While allocating components by roles, several components may be grouped. Rule GroupClassDiagrams in Figure 5.20 realizes the grouping of classes in the test model. For that, it extends two transformation rules called SetNestedClasses and SetExternalInterface4ClassDiagrams (Lines 2–4). The former enables the setting of the system classes as nested classes of the test class. The latter sets external interfaces by comparing the name of the interfaces (Figure 5.21). By name matching, the interface is internal. Otherwise, they are communicating with the environment (Lines 4–6). By means of Rule

Description: Role assignment for the SUT class: According to the Test Directives model, check whether the new class is meant to be assigned to an SUT, if yes, then assign the stereotype <<SUT>> to the class.

```
1  RULE RoleAssignment_Classes(uml_element, u2tp_element)
2      FORALL ::testdirectives::GroupingClasses@TestDirSrc td_element
3      WHERE IF td_element.groupingPurpose = SUT
4                THEN u2tp_element.appliedStereotypes = SUT
5                // Assign the stereotype <<SUT>> to u2tp_element
6            ELSEIF td_element.groupingPurpose = TestComponent
7                THEN u2tp_element.appliedStereotypes = TestComponent
8                // Assign stereotype <<TestComponent>> to u2tp_element
9            ELSE TRUE
10           ENDIF
11     MAKE ::uml2::Class@U2TPtgt u2tp_element
12     SET u2tp_element.name = td_element.newTestClassName
13           // Set the name of the test element
14     LINKING ClassRoles4SUT
15         WITH theUMLClass = uml_element,
16              theU2TPSUT = u2tp_element
17 ;
```

Figure 5.19: Role Assignment for Test Component and SUT Classes

Description: Group UML 2.0 class elements together to one test class according to the grouping criteria which is specified in the Test Directives model.

```
1  RULE GroupClassDiagrams(uml_element1, uml_element2, u2tp_element)
2      EXTENDS SetNestedClasses(uml_element1, uml_element2, u2tp_element),
3              // Nesting system classes into the test class
4      SetExternalInterface4ClassDiagrams(uml_element1,uml_element2,u2tp_element)
5              // Remove the internal interfaces and
6              // set the interfaces to the environment
7      FORALL ::uml2::Class@UMLsrc uml_element1,
8              :: uml2::Class@UMLsrc uml_element2,
9              :: testdirectives :: GroupingClasses@TestDirSrc td_element
10     WHERE td_element.theClass.name = uml_element1.name
11           // Compare testdirective class with UML class
12     OR td_element.theClass.name = uml_element2.name
13     AND CheckEqualInterface4ClassDiagrams(uml_element1, uml_element2)
14         // Check whether the classes share the same interface
15     MAKE ::u2tp::Class@U2TPtgt u2tp_element
16 ;
```

Figure 5.20: Grouping Classes

GroupClassDiagrams only classes specified in the Test Directives model are grouped by comparing the names of the system classes and verifying their

Description: Set the new ports for the grouped UML 2.0 elements. This rule is needed during the grouping mechanism with multiple UML 2.0 classes. For the grouping, all ports and interfaces between the grouped classes will be removed but the ports and interfaces to the environment are still kept.

```
1  RULE SetExternalInterface4ClassDiagrams(uml_class1, uml_class2, u2tp_class)
2      FORALL ::uml2::Class uml_class1,
3              :: uml2::Class uml_class2
4      WHERE uml_class1.nestedClassifier.nestedInterface.name
5            != uml_class2. nestedClassifier . nestedInterface . name{}
6                -> interfaces{} // Retrieve the external interfaces
7      MAKE ::u2tp::Class@U2TPtgt u2tp_class
8      SET u2tp_class.nestedClassifier . nestedInterface . name = interfaces
9          // Set the external interfaces of the system
10         // model classes to the test class
11 ;
```

Figure 5.21: Set External Interfaces

interfaces (Lines 10–12 in Figure 5.20). Only the external interfaces are adopted to the test model and the internal ones are neglected.

Figure 5.22 shows the application of the mentioned rules to the case study. The UML 2.0 diagram on the upper left side of the figure originates from the system model. Herein, all classes and interfaces between the Bluetooth

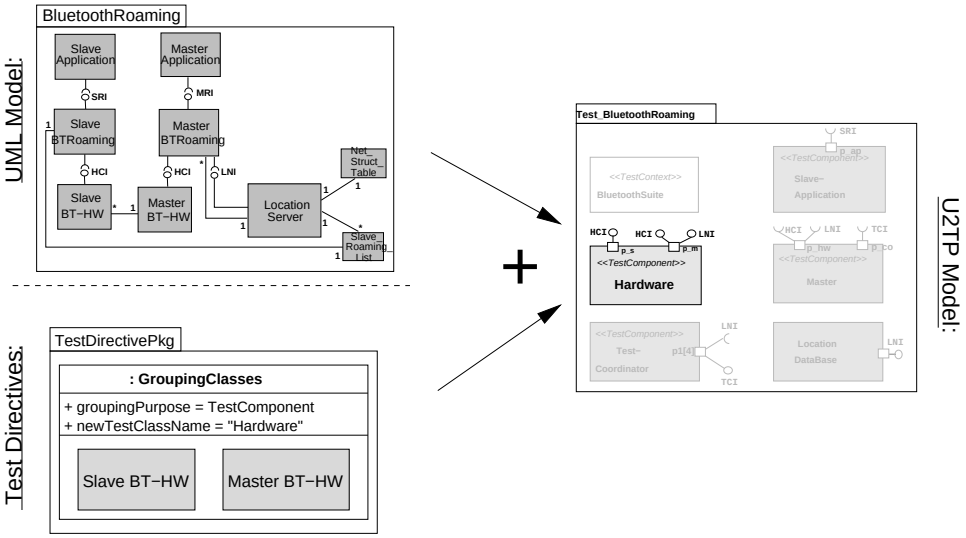


Figure 5.22: Assigning the Bluetooth Hardware to Test Component

classes are depicted. In order to group the Bluetooth classes Slave BT-HW and Master BT-HW to one test component the Test Directives model on the lower left side of the figure is needed. To do so, the grouping purpose (i.e. test component) is indicated in the Test Directives model class. The new name of the test class is specified as Hardware. Together with the Test Directives model and the transformation rules, the system model is transformed to the target U2TP test model as shown on the right side of Figure 5.22.

Test Context In U2TP, a test context class incorporates all test cases with the same initial test configuration. Rule ListTestCasesInTestContext in Figure 5.23 defines the transformation rule to list all the test cases. It first creates a new test context class and applies stereotype <<TestContext>> to it and selects existing system scenarios specified in diverse behavior diagrams for test case specification (Lines 2–3). Herefore, a Test Directives model is required to choose appropriate system scenarios for test case modeling (Line 12–14 in Figure 5.24). From these scenarios, test cases can be derived. They are assigned by new names by retrieving the system scenario names and append the prefix "Test" to the system scenario names (Lines 13–14).

Test cases are retrieved from existing Interaction Diagrams which specify test cases. The relationship between test cases and the appropriate diagrams are set in the tracking relation called TestCaseFromInteraction (Lines 4–6 in Figure 5.23). The names of the test cases are listed as operations within the test context class (Line 8). Since a test case returns a verdict as its result,

Description: List existing test cases as operations in the test context class. Assign test cases with stereotype <<TestCase>>.

```
1  RULE ListTestCasesInTestContext(scenario, testcase, testcontext)
2      EXTENDS CreateTestContext(testcontext),
3              AssignNameToTestCases(scenario, testcase)
4      WHERE TestCaseFromInteraction
5          LINKS theInteraction = interaction,
6                theTestCase = testcase
7      MAKE ::u2tp::TestContext@U2TPtgt testcontext
8      SET testcontext.ownedOperation.name = testcase.name,
9          // Test cases are listed as operations in test context class
10         testcontext.ownedOperation.type = Verdict,
11         // Testcases always return test verdicts
12         testcontext.ownedOperation.appliedStereotypes = TestCase
13         // Assign stereotype <<TestCase>> to the operations
14 ;
```

Figure 5.23: Listing Test Cases in Test Context

Description: Assign names to test cases in Interaction Diagrams or State Machines. Look into the ChooseTestCase class in the Test Directives model. Compare the name of the diagrams with the name in the ChooseTestCase class of the Test Directives model. Assign the name of the new test case with "Test" + name of the scenario.

```
1  CLASS TestCaseFromInteraction {
2      ::uml2::Interaction theInteraction;
3      ::u2tp::TestCase theTestCase;
4  };

5  CLASS TestCaseFromStatemachine {
6      ::uml2::StateMachine theStatemachine;
7      ::u2tp::TestCase theTestCase;
8  };

9  RULE AssignNameToTestCases(scenario, testcase)
10     FORALL ::uml2::Interaction@UMLsrc interaction,
11            ::uml2::StateMachine@UMLsrc statemachine,
12            :: testdirectives ::ChooseTestCases@TestDirSrc choice
13     WHERE interaction.name = choice.name
14     AND statemachine.name = choice.name
15         // Only take the interactions, statemachines and acitivity
16         // diagrams which are chosen in the testdirective model
17     MAKE ::u2tp::TestCase@U2TPtgt testcase
18     SET testcase.name = append("Test", interaction.name),
19         testcase.name = append("Test", statemachine.name)
20         // Give the test cases a name by the prefix "Test".
21     LINKING TestCaseFromInteraction
22         WITH theInteraction = interaction,
23              theTestCase = testcase
24     LINKING TestCaseFromStatemachine
25         WITH theStatemachine = statemachine,
26              theTestCase = testcase
27 ;
```

Figure 5.24: Test Case Selection and Naming

the return type of the test cases are set to Verdict (Line 10). At last, the <<TestCase>> stereotype is applied to the test cases (Line 12). Figure 5.25 shows the test context of the Bluetooth case study after the application of Rule ListTestCaseInTestContext. There, a new test context class is created with both test cases TestRoaming_DataSending and TestRoaming_noWarning listed as operations (Line 8).

Test Configuration A test configuration defines the topology of the test system and the SUT. For its specification, Composite Structure Diagrams provide a good means since it allows the specification of ports. To achieve

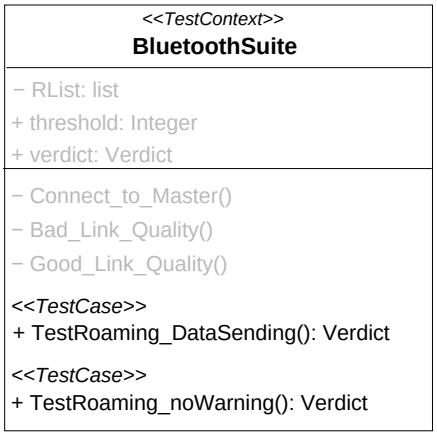


Figure 5.25: A Test Context Class for the Bluetooth Test Model

this, system classes are grouped in a Composite Structure Diagram. Internal connections between the grouped classes are omitted and only the external connections to the environment are adopted. Rule CheckExternalInterface4CompositeStructureDiagrams in Figure 5.26 checks whether a port connects with the environment. By definition, ports may own required and

Description: Check whether the ports between classifiers in the composite structure diagram realy communicate with each other via interfaces. Classifiers in a composite structure diagram have ports which are connected via interfaces. If the ports are behavior ports, then check whether the interfaces of the classifiers correspond to each other, i.e. uml_encl1 provides the interface and uml_encl2 requires this interface.

```
1  PATTERN CheckExternalInterface4CompositeStructureDiagrams
2                                              (uml_encl1, uml_encl2)
3      FORALL ::uml2::EncapsulatedClassifier uml_encl1,
4              ::uml2::EncapsulatedClassifier uml_encl2
5              // Classes have ports which are connected via interfaces
6      WHERE uml_encl1.ownedPort.isBehavior = true
7            // behavior port
8          AND uml_encl2.ownedPort.isBehavior = true
9          AND uml_encl1.ownedPort.required != uml_encl2.ownedPort.provided
10         OR uml_encl2.ownedPort.required != uml_encl1.ownedPort.provided
11         // uml_encl1 provides the interface and uml_encl2 requires
12         // the interface at the ports or vice versa
13 ;
```

Figure 5.26: Check External Interfaces

provided interfaces which connect classes to each other. Thus, in case of internal ports, their interface methods must correspond. By checking the inequality of the interface methods, external ports can be recognized and internal ports neglected (Line 9–10).

Description: Set the new ports for the grouped UML classifiers in the Composite Structure Diagram. This rule is needed during the grouping mechanism with multiple UML 2.0 classifiers. For the grouping, all ports between the grouped elements will be removed but the external ports to the environment are still kept.

```
1  RULE SetExternalInterface4CompositeStructureDiagrams
2                                  (uml_encl1,uml_encl2,u2tp_encl)
3      FORALL ::uml2::EncapsulatedClassifier uml_encl1,
4             :: uml2::EncapsulatedClassifier  uml_encl2
5                        // EncapsulatedClassifiers  own ports
6      WHERE
7      CheckExternalInterface4CompositeStructureDiagrams(uml_encl1, uml_encl2)
8      MAKE ::u2tp::EncapsulatedClassifier@U2TPtgt u2tp_encl
9      SET u2tp_encl.ownedPort.required = uml_encl1.ownedPort.required,
10         u2tp_encl.ownedPort.provided = uml_encl1.ownedPort.provided,
11         // Copy the external interfaces  from uml_encl1 to u2tp  class
12         u2tp_encl.ownedPort.required = uml_encl2.ownedPort.required,
13         u2tp_encl.ownedPort.provided = uml_encl2.ownedPort.provided
14         // Copy the external interfaces  from uml_encl2 to u2tp  class
15 ;
```

Figure 5.27: Set External Interfaces for Grouped Components

Description: Connect ports between parts in test configuration. After the grouping of classes or instances in the test configuration, the ports between the test components or/and SUT must be connected to each other via ports.

```
1  RULE ConnectingPorts(uml_sc1, uml_sc2)
2      FORALL ::uml2::StructuredClassifier uml_sc1,
3             :: uml2:: StructuredClassifier  uml_sc2
4      WHERE
5        CheckExternalInterface4CompositeStructureDiagrams(uml_sc1, uml_sc2)
6      SET uml_sc1.ownedConnector.end.partWithPort
7          = uml_sc2.ownedConnector.end.partWithPort
8          // Connect the ends of the connector to
9          // the  ports  of  uml_sc1 and uml_sc2.
10 ;
```

Figure 5.28: Connect Ports in the Test Configuration

After having removed the internal ports of a test component, external ports are adopted to the test configuration by Rule SetExternalInterface4Composite-StructureDiagrams (Figure 5.27). The connection between test components and the SUT is realized in Rule ConnectingPorts (Figure 5.28) which connects both ends of a connector with the ports (Lines 6–7).

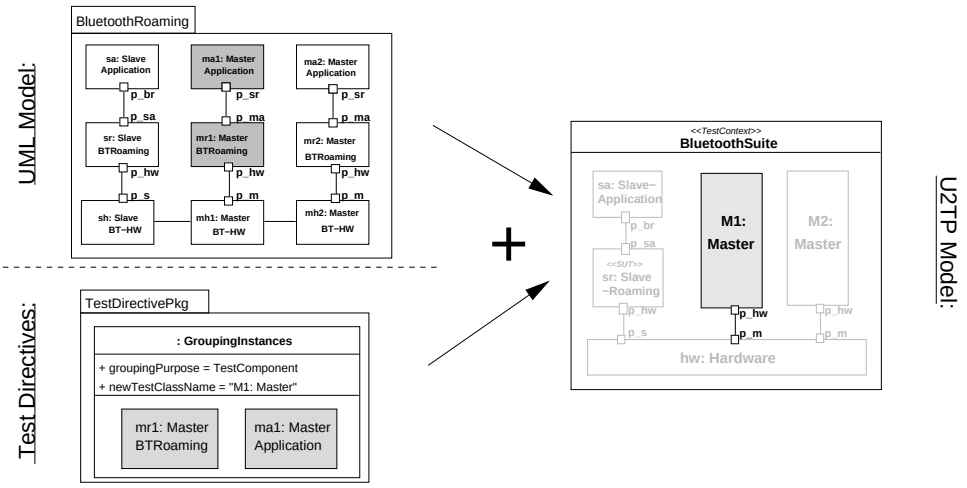


Figure 5.29: Bluetooth Test Model Configuration

Figure 5.29 shows the test configuration of the Bluetooth roaming case study after transformation. In the system configuration, one Bluetooth Slave and two Bluetooth Masters are shown. The Test Directives model specifies the grouping of instance mr1 and instance ma1 of Master M1. The resulting component is a test component named M1 of type Master. Consequently, mr1 and ma1 are combined and their internal ports p_sr and p_ma are not adopted into the test model. In the resulting test configuration, the application and roaming components of the masters are merged. Port p_hw is identified as an external port and is thus connected with the underlying hardware component.

Test Control Test control cannot be derived from existing system diagrams. It must be created and specified from the scratch. To do so, Rule CreateTestControl in Figure 5.30 creates an Interaction Diagram with interactions referencing to the test cases to be executed in the test (Line 4). The name of the test control derives from the system model name with suffix "_TestControl" (Line 7). Rule ConcatenateTestCases4TestControl in Figure 5.31 retrieves relevant test cases (Lines 7–8) and concatenates those in the test control diagram (Line 11). Decisions are made considering test results. Usually, if the result of a test case is pass, further test cases are executed. In case that the test verdict of the last test cases turns out to be fail,

Description: Create a test control class. Test control is best specified as Interaction Diagram, whereas interactions are referenced as test cases in the test control.

```
1  RULE CreateTestControl(testcontrol, testcase)
2      WHERE U2TPModel2UMLModel
3          LINKS theUMLModel = systemModel, theU2TPModel = testModel
4      MAKE ::uml2::Activity@UMLsrc testcontrol
5          // Interaction Overview Diagrams specialize Activity Diagrams.
6          // Create an Activity Diagram for test control.
7      SET testcontrol.name = append(testModel.name, "_TestControl")
8          // Assign name for the test control class
9  ;
```

Figure 5.30: Create Test Control Class

the whole test is stopped immediately (Line 9 and 14). The specification of the termination of the test is realized by defining the final node (Line 15).

Figure 5.32 illustrates the test control class of the Bluetooth roaming case study with the execution of both test cases TestRoaming_DataSending and TestRoaming_noWarning. The second test case will be executed only if the previous passes. Otherwise the whole test terminates.

Description: Concatenate test cases together for the definition of test control, whereas interactions are referenced as test cases in the test control. Test control is specified in an Interaction Diagram.

```
5  RULE ConcatenateTestCases4TestControl(testcontrol, testcase)
6      EXTENDS CreateTestControl(testcontrol, testcase)
7      WHERE TestCaseFromInteraction
8          LINKS theInteraction = interaction, theTestCase = testcase
9      SET testcontrol.node.decisionInput = "verdict==pass",
10         // Decision definition for true case within the interaction flow.
11         testcontrol.InteractionOccurrence.refersTo = testcase,
12         // Testcases are referenced fragments in Interaction Overview Diagram.
13         // Concatenate the testcases together.
14         testcontrol.node.decisionInput = "verdict==fail",
15         testcontrol.node = FinalNode
16         // Decision definition for false case within the interaction flow.
17     LINKING TestControl2TestCases
18         WITH theTestControl = testcontrol, theTestCases = testcase
19 ;
```

Figure 5.31: Concatenate Test Cases in Test Control

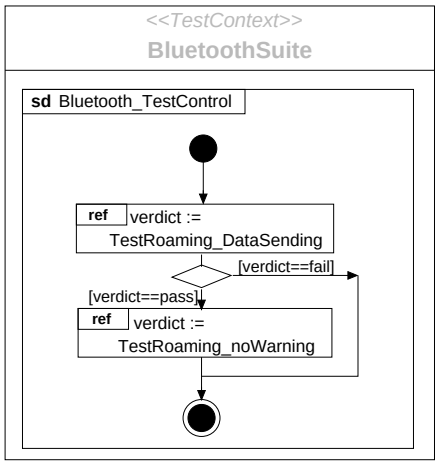


Figure 5.32: A Test Control Diagram for the Bluetooth Test Model

5.3.2.3 Transformation Rules for Test Behavior

The behavior of a U2TP model is specified in test cases and defaults where communication between test system components and the SUT are defined. To achieve that, scenarios in the system model can be adopted for test case and default specifications. Modification on the configuration must be made according to the test configuration defined in the test architecture. Default behavior for components and messages can be generated automatically.

Test Case Test cases can be derived from system scenarios. To do so, Interaction Diagrams are optimal to describe communication between components whereas State Machines provide good means to model the internal behavior of a single component.

For modeling test cases with Interaction Diagrams, system scenarios are selected and all the Interaction Diagram elements, i.e. lifelines, messages, fragments and gates, are adopted to the test model (Line 2 in Figure 5.33). The roles of the lifelines must correspond to the component roles defined in the test configuration. To achieve this, Rule AssignSUTLifelineInTestCase references to tracking class TestCaseFromInteraction showing the relationship between a test case and an Interaction Diagram (Lines 6–7) and thus only utilizes the Interaction Diagrams selected for test case specifications. Subsequently, stereotype <<SUT>> is applied to the SUT lifeline in the diagram (Line 14)[12]. According to the U2TP standard, test component are

[12]In this rule, SUT lifelines are grouped to a single lifeline. According to the U2TP standard, multiple SUT lifelines can be defined in a U2TP model.

not stereotyped for lifelines[13]. Therefore, no such transformation rule for assigning test component stereotype is needed.

Description: Assign SUT lifeline in the Interaction Diagrams. Copy the system scenario to the test model as test case diagram.

```
1  RULE AssignSUTLifelineInTestCase(umlInteraction, u2tpInteraction)
2      EXTENDS CopyUMLInteractionToU2TP(umlInteraction, u2tpInteraction)
3      FORALL ::testdirectives::GroupingInstances@TestDirSrc td_element
4      // The Test Directives model encorporates information
5      // about the SUT instance
6      WHERE TestCaseFromInteraction
7          LINKS theInteraction = umlInteraction, theTestCase =u2tpInteraction
8          // Get the design scenario and the test cases
9          // in the Interaction Diagrams
10     AND td_element.groupingPurpose = SUT
11     AND td_element.theInstance = theSUTInstance
12         // Look into the Test Directives model which
13         // instance should be the SUT instance
14     SET theSUTInstance.appliedStereotypes = SUT
15         // Assign stereotype <<SUT>> to the SUT instance
16 ;
```

Figure 5.33: Assign SUT Lifeline in Interaction Diagram

After creating a test case and assigning the SUT lifeline in the test case, other lifelines must be grouped according to test configuration. There, internal communication between the grouped lifelines must be removed and external communication to their adjacent lifelines need to be recognized.

The grouping of lifelines is implemented by Rule GroupingLifelineInTestCase in Figure 5.34. Firstly, external messages between the grouped lifelines and their adjacent lifelines are detected by Rule SetExternalCommunication4Lifelines in Figure 5.35. A communication message between grouped lifelines is considered external if it is sent by one lifeline and is not received by the other lifeline (Line 5–9). External messages are kept in the test scenario. When grouping the lifelines, a Test Directives model is needed to select the appropriate lifelines according to test configuration specification (Lines 8–9, Figure 5.34). A prerequisite for grouping lifelines is that internal messages must exist. Thus, the transformation rule assures that internal communication exist between the chosen lifelines and names the new lifeline according to the Test Directives specification (Line 13 and 16).

[13]In U2TP, the SUT extends UML 2.0 meta-class Property, while test component extends meta-class Classifier. According to UML 2.0, properties need stereotyping, classifiers not.

Description: Grouping of lifelines according to the specification of Test Directives model for test components and SUT. Delete the internal communication between the lifelines.

```
1  RULE GroupingLifelineInTestCase(umlLifeline1, umlLifeline2, u2tpLifeline)
2      EXTENDS SetExternalCommunication4Lifelines
3                  (umlLifeline1, umlLifeline2, u2tpLifeline)
4              // Determine the communication to other lifelines.
5      FORALL ::uml2::Lifeline umlLifeline1,
6              ::uml2::Lifeline umlLifeline2,
7              :: testdirectives ::GroupingLifelines@TestDirSrc td_element
8      WHERE td_element.theLifeline.name = umlLifeline1.name
9          OR td_element.theLifeline.name = umlLifeline2.name
10             // Compare the names of the lifelines defined
11             // in the testdirective model which should be grouped together
12             // with the names of the lifelines in the system scenario.
13         AND CheckInternalCommunication4Lifelines(umlLifeline1, umlLifeline2)
14             // Assure that internal events between the lifelines exist.
15     MAKE ::uml2::Lifeline u2tpLifeline
16     SET u2tpLifeline.name = td_element.newLifelineName
17         // Set the new name of the grouped lifeline according to the
18         // Test Directives model.
19 ;
```

Figure 5.34: Grouping of two Lifelines

Description: Define the communication message to the adjacent lifeline in an Interaction Diagram after the grouping of the instances to test components and SUT.

```
1  RULE
2   SetExternalCommunication4Lifelines(umlLifeline1,umlLifeline2,u2tpLifeline)
3      FORALL ::uml2::Lifeline umlLifeline1,
4              ::uml2::Lifeline umlLifeline2
5      WHERE umlLifeline1.interaction.message.receiveEvent
6            != umlLifeline2.interaction.message.sendEvent{} -> receiveEvents{}
7            // Only external receiving events to the neigbouring lifeline
8      OR umlLifeline2.interaction.message.receiveEvent
9         != umlLifeline1.interaction.message.sendEvent -> sendEvents{}
10        // Only external sending events to the neigbouring lifeline
11     MAKE ::uml2::Lifeline u2tpLifeline
12     SET u2tpLifeline.interaction.message.receiveEvent = receiveEvents,
13         u2tpLifeline.interaction.message.sendEvent = sendEvents
14         // Set the sending and receiving events to the neighbouring lifeline
15 ;
```

Figure 5.35: Set External Communication Message between Lifelines

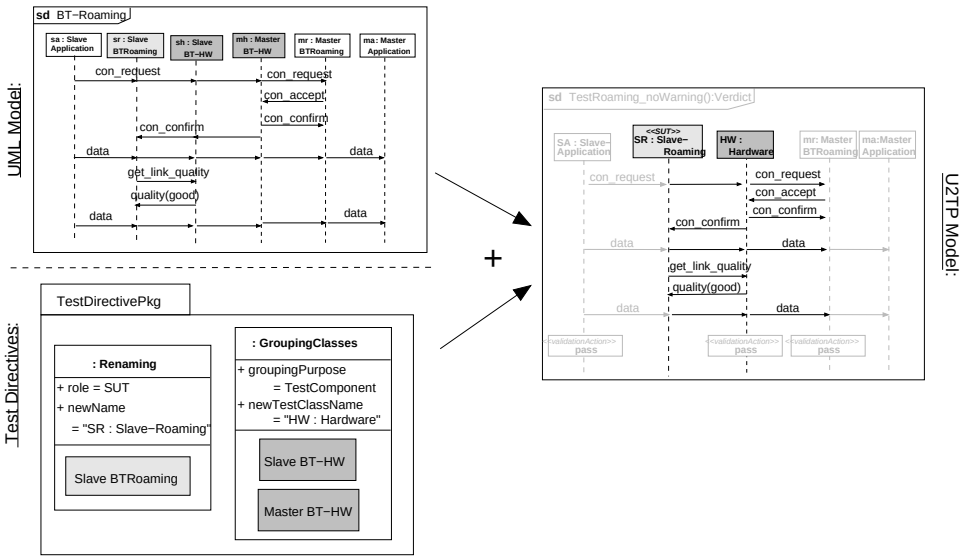


Figure 5.36: Grouping Lifelines for the Bluetooth Test Scenario

Figure 5.36 exemplifies the application of the transformation Rules AssignSUTLifelineInTestCase and GroupingLifelineInTestCase. In the system model on the left upper corner, six lifelines are shown representing the different layer instances of a Bluetooth Slave and a Bluetooth Master. According to the specification of GS1 in the Test Directives model, the lifeline of class Slave BTRoaming should be assigned to SUT and named as SR. Additionally, the lifelines of classes Slave BT-HW and Master BT-HW should be grouped. Thus, the internal messages between these two lifelines, i.e. con_request, con_confirm and data are removed from the test scenario and the new test component is named to HW.

General Default Behavior Default is a test-specific concept which cannot be derived from the system model. Thus, refinement on the model must be done and new diagrams for defaults be created. To describe the local behavior of a component by states and transitions State Machines provide good means.

If any unexpected response from the SUT is caught by a test component independently from its current state, then a *general default* is needed. To do so, Rule DefineDefaultInStateMachine in Figure 5.37 creates a State Machine for each test component of the test system. The general default diagram consists of a single ANY state with one transition receiving an ANY event (Lines 11–16). The test verdict is set to inconclusive (Line 17).

Description: General default specification in State Machines with any states and any input message. Set test verdict to inconclusive.

```
1  CLASS State4Statemachine{
2      ::uml2::StateMachine theStatemachine;
3      ::uml2::State theState;
4  };

5  RULE DefineDefaultInStatemachine(selfElement, umlStateMachine)
6      EXTENDS NamingDefault(selfElement, umlStateMachine)
7      MAKE ::uml2::StateMachine umlStateMachine,
8           ::u2tp::LiteralAny U2TPAny
9      SET umlStateMachine.region.state = U2TPAny,
10         // Make an ANY state in the statemachine
11         umlStateMachine.region.transition.source
12                         = umlStateMachine.region.state,
13         umlStateMachine.region.transition.target
14                         = umlStateMachine.region.state,
15         // The transition has the ANY state as source and final state
16         umlStateMachine.region.transition.trigger = U2TPAny,
17         umlStateMachine.region.transition.guard = setVerdict("inconc")
18         // Define an transition with ANY input
19         // and setverdict(inconc) as output
20     LINKING State4Statemachine
21         WITH theStatemachine = umlStateMachine, theState = U2TPAny
22 ;
```

Figure 5.37: Specifying General Default Behavior in State Machines

Component-Based Default Behavior In U2TP, default behavior can be specified in different scopes either attached to a component, a message, or a state etc. If a default is attached to a component, all unexpected messages sent to this component are examined and evaluated. Rule DefineDefault4Component in Figure 5.38 extends the State Machine created for the general default specification and adds new transitions for the *component-based default* behavior specification (Lines 3–5). Line 6 shows the creation of the ANY parameter for a message. The definition of the new transition starts from the ANY state and receives a certain message with the ANY value as its parameter (Lines 7 and 13). Finally, the default goes to its final state and the test verdict is set to fail (Lines 9 and 17).

Figure 5.39 shows the specification of a general default and a component-based default for the Bluetooth example. The component-based default is applied to the Hardware test component. According to test cases TestRoaming_DataSending and TestRoaming_noWarning, the messages con_request and get_Link_Quality can be sent from SUT to the hardware test component. Thus, when receiving these messages with wrong parameters than specified in the test cases, the test fails and the hardware test component terminates.

Description: This rule is used for a component-based default specification. Whenever a default is attached to a component, take the behavior diagram into account, e.g. Interaction Diagram. Specify a default behavior for all messages which are sent to the (test) component in the Interaction Diagram. The default behavior is specified as a State Machine. Take these sending messages and specify these in the default State Machine and assign test verdict "fail" as output.

```
1  RULE DefineDefault4Component(theMessage)
2      EXTENDS CheckReceivedMessages(theMessage)
3      WHERE State4Statemachine
4          LINKS theStatemachine = umlStateMachine,
5                theState = U2TPAnyState
6      MAKE ::u2tp::LiteralAny theAnyParameter
7      SET umlStateMachine.region.state = U2TPAnySate,
8          // Take the existing ANY state
9          umlStateMachine.region.transition.target
10                             = umlStateMachine.region.FinalState,
11         // The target state is the final state
12         umlStateMachine.region.transition.trigger
13                             = theMessage(theAnyParameter),
14         // Define a transition with theMessage which is sent
15         // from SUT to a test component as input and any
16         // unexpected parameter as output
17         umlStateMachine.region.transition.guard = setverdict(" fail ")
18         // If message with any other parameters are received,
19         // the test verdict is fail.
20 ;
```

Figure 5.38: Specify Component-Based Defaults

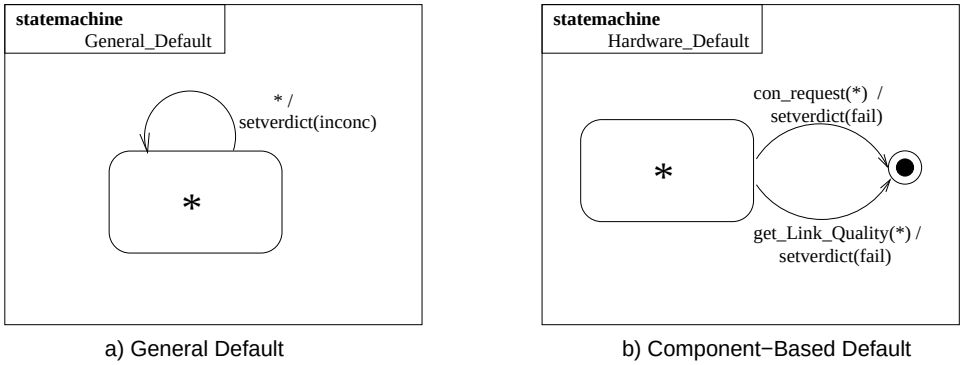


Figure 5.39: Specifying General and Component-Based Defaults

Message-Based Default Behavior Despite of the general and component-based default behavior, a default can also be specified for more fine-grained scopes where default is activated for a certain message. Rule DefineDefault4SpecialMessages in Figure 5.40 defines a *message-based default* by means of a Test Directives model. However, the Test Directives provides information considering the start and end states of a transition, the input and output messages, the lifeline of the message and the verdict value (Lines 13–21).

Figure 5.41 shows the specification of a message-based default for the test scenario shown on left upper corner of the figure[14]. According to the Test Directives, a default behavior is triggered by message con_confirm of Test Coordinator CO attached to the ANY state. The transition ends at the final node of the State Machine. As a consequence, whenever Test Coordinator receives a con_confirm with an unexpected parameter, the default is invoked, the test verdict is set to fail and the behavior of Test Coordinator terminates.

Description: Specify default behavior for a special message which is sent from SUT to the test component within a Interaction Diagram. The default behavior is specified as a State Machine. Information about the default behavior is specified in the test directive model: name of the message, the verdict, start and end state of the transition within the State Machine.

```
5  RULE DefineDefault4SpecialMessage(message)
6      EXTENDS CheckReceivedMessages(message)
7      FORALL
8          :: testdirectives :: DefaultForMessages@TestDirSrc td_MessageDefault
9      WHERE State4Statemachine
10         LINKS theStatemachine = umlStateMachine,
11               theState = U2TPAnyState
12     MAKE ::u2tp::LiteralAny theAnyParameter
13     SET umlStateMachine.region.state = td_MessageDefault.StartState,
14         // Take the start state specified in the testdirective model
15         umlStateMachine.region.transition.target = td_MessageDefault.EndState,
16         // End state of the transition
17         umlStateMachine.region.transition.trigger
18               = td_MessageDefault.theMessage(theAnyParameter),
19         // Define a transition with theMessage as input
20         umlStateMachine.region.transition.guard
21                     = setverdict(td_MessageDefault.theVerdict
22         // Set the test verdict according to the Test Directives model.
23     LINKING Default4Message
24         WITH theDefault = umlStateMachine, theMessage = message
25 ;
```

Figure 5.40: Specify Message-Based Defaults

[14]The test scenario shows only an excerpt of the test case.

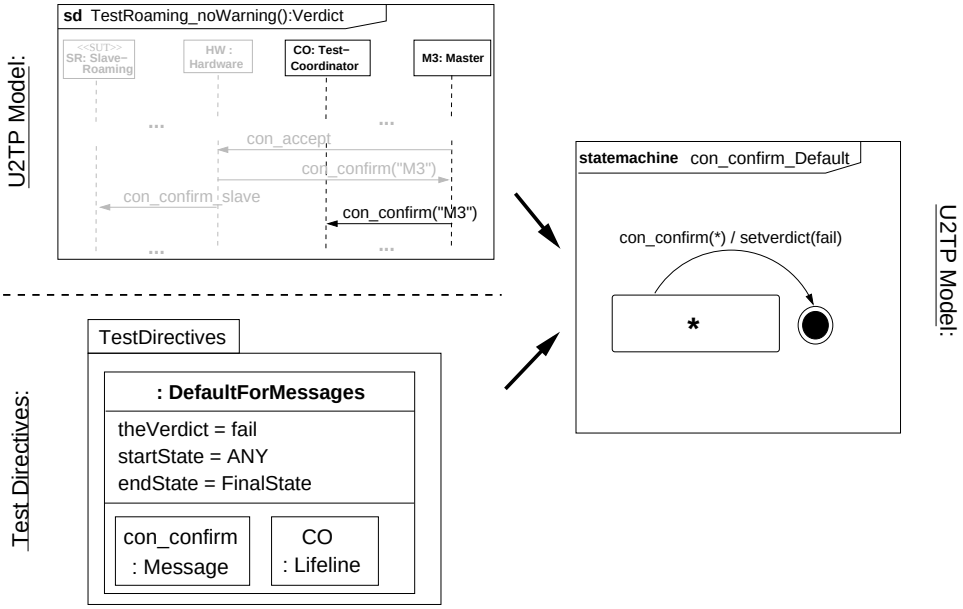


Figure 5.41: Specification of Message-Based Default

Description: Attach default to a message The notation of a message-based default is a comment attached to the message.

```
1  RULE AttachDefault2Message(message, default)
2      EXTENDS NamingDefault(message, default)
3              // Assign the name of the default
4      FORALL ::uml2::Lifeline@UMLsrc theLifeline,
5        :: testdirective :: DefaultForMessages@TestDirSrc td_DefaultForMessages
6      WHERE Default4Message
7          LINKS theDefault = default, theMessage = message
8          // Get the default specification and
9          // the message to which the default is attached to
10     AND td_DefaultForMessages.theLifeline = theLifeline
11         // Retrieve the lifeline with the message
12         // where the default should be attached to.
13         // The name of the lifeline is defined in the testdirective model.
14     SET theLifeline.interaction.message.ownedComment.body
15                         = append("default ", default.name)
16         // Attach the default to the message of the lifeline
17         // The notation of a message-based default is the comment symbol
18 ;
```

Figure 5.42: Specify Default to a Message

Activate Defaults However, simply specifying the default behavior is not enough. In order to be able to use a default, it must be activated by attaching to the message in the test case behavior. This is realized by Rule AttachDefault2Message shown in Figure 5.42. Herefore, the tracking class Default4Message retrieves the default behavior and its appropriate message (Lines 6–7). In addition, the lifeline of the component is filtered out by means of the Test Directives model (Line 10). In U2TP, the notation of a message-based default is depicted by a comment symbol attached to the message referencing to the default specification (Lines 14–15). Figure 5.43 shows an excerpt of test case TestRoaming_noWarning activating the con_confirm_Default modeled in Figure 5.41.

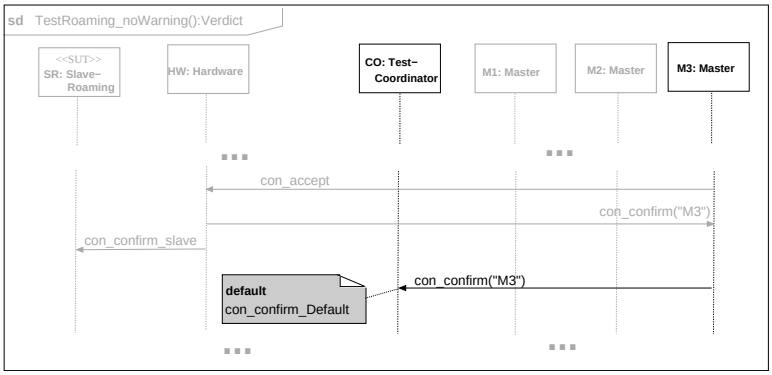


Figure 5.43: Attach Default to a Message

5.3.2.4 Transformation Rules for Test Data

The simplest way to specify test data is to adopt all data from the system model. In this way, it can be assured that data exchanged between the test system and the SUT also exist in the test model, even though other data might exist which are not relevant for testing, e.g. internal communication messages of grouped components.

Data Pool Figure 5.44 shows Rule CopyDataToTestModel copying data types from the system model, including their names, attributes and operations (Lines 8–10). An alternative to copying is to import the whole system data package to the test model, assuming that system data is stored in a separated package. Figure 5.45 shows the adoption of test data types using Rule CopyDataTypeToTestModel. The system model on the left side incorporates four data types con_request_Type, con_confirm_Type, roamingList_Type, data and two data instances. The data types are adopted to test model including their possible attributes and operations.

Description: Adopt data from the system model.

```
1  RULE CopyDataTypeToTestModel(umlDataType, u2tpData)
2      FORALL ::uml2::DataType umlDataType
3      WHERE umlDataType.name = name
4          AND umlDataType.ownedAttribute = ownedAttribute
5          AND umlDataType.ownedOperation = ownedOperation
6          // Get all features of data types in the system model
7      MAKE ::u2tp::DataPool u2tpData
8      SET u2tpData.$name = umlDataType.$name,
9          u2tpData.$ownedAttribute = umlDataType.$ownedAttribute,
10         u2tpData.$ownedOperation = umlDataType.$ownedOperation
11         // Copy all features of the system model's data types
12 ;
```

Figure 5.44: Adopt Data from System Model

Data partitions and data selector cannot be retrieved from the system model. Since these concepts are not mandatory, no transformation rules are specified. Also, coordination data between test components might be needed which cannot be derived from the system model. They must be included manually into the test model during the refinement step.

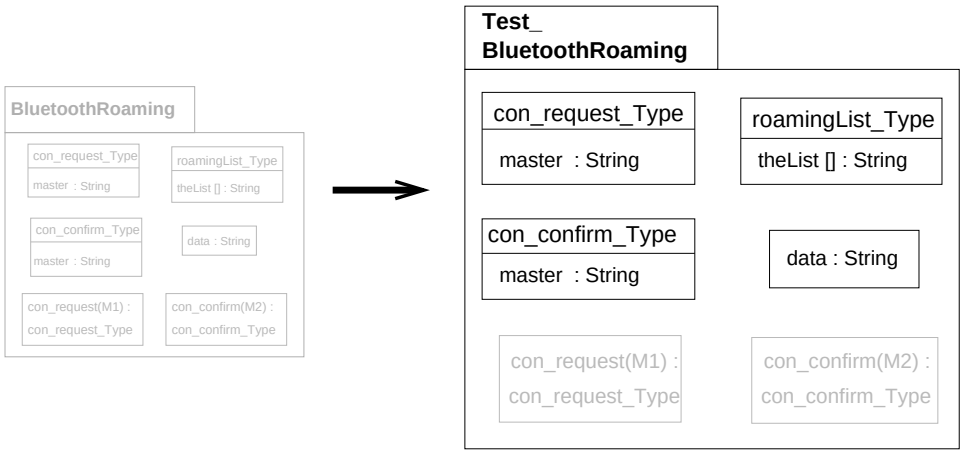


Figure 5.45: Adopting Test Data

5.3.2.5 Transformation Rules for Time

Timers are test-specific concepts which cannot be derived from the system model directly. In our transformation, we implemented rules to define a test component timer and a guarding timer.

Test component Timer A *test component timer* restricts the duration of the test component behavior for a certain test case execution. Herefore, a timer is added to the test component before the test behavior starts and is stopped at the end of the test behavior before the test componet sets the test verdict.

Description: Add timer to a message of a certain lifeline in an Interaction Diagram. The Test Directives model specifies where a timer should be attached to and also to which instance the timer should be attached to.

```
1  CLASS Timer4Lifeline {
2      ::uml2::Lifeline   theLifeline ;
3      ::u2tp::Timer theTimer;
4  };
```

```
1  RULE AddTestComponentTimer()
2      FORALL ::uml2::Lifeline@UMLsrc theLifeline,
3              ::uml2::Message@UMLsrc theMessage
4      WHERE CheckRoleForLifeline(TestComponent, theLifeline)
5          // Take only the test component lifelines for
6          // the test component timer specification
7      AND SearchStartBehavior(theLifeline,theMessage)
8          // Look for the behavior beginning
9      AND SearchEndBehavior(theLifeline, theValidationAction)
10         // Look for the behavior end
11     MAKE ::u2tp::timer theTimer
12         // Create a timer in the test case
13     SET theTimer.name = append("Timer_", theLifeline.name),
14         // Set the name of the timer
15         theLifeline .coveredBy.toBefore.theMessage = theTimer.start("100s"),
16         // Add the start timer operation and attach the timer at the
17         // beginning of the test case. The time duration is set to
18         // an accordingly big time unit.
19         theLifeline .coveredBy.toBefore.theValidationAction=theTimer.stop()
20         // Attach the stop timer operation before the validation action
21     LINKING Timer4Lifeline
22         WITH theLifeline = theLifeline,
23              theTimer = theTimer
24 ;
```

Figure 5.46: Adding Test Component Timer to Test Scenario

Rule AddTestComponentTimer in Figure 5.46 realizes the specification of a test component timer. To do so, lifelines of test components are first searched (Line 4)[15]. Afterwards, a timer is created and named with the prefix "Timer_" followed by the name of the test component. Herefore, the expiration duraion of the test component timer should be calculated

[15]The behavior of the SUT should not be manipulated.

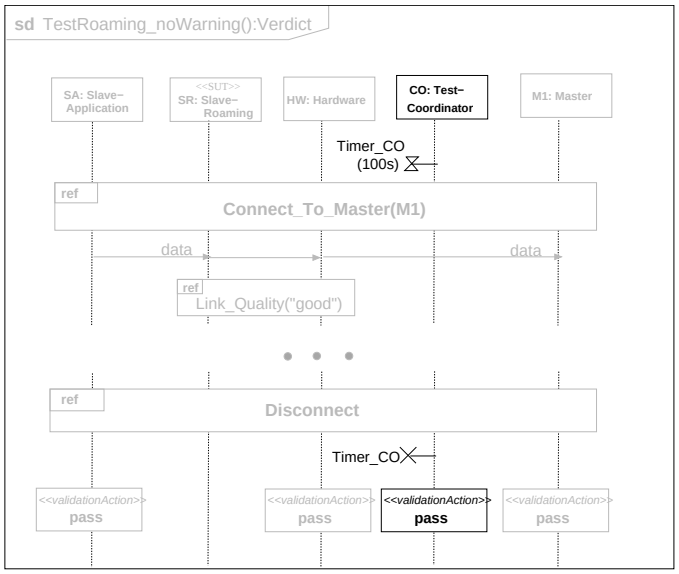


Figure 5.47: Defining Test Component Timer

generously to allow the test component to finish its test behavior. Therefore, a timer duration of 100 seconds is defined and attached in the beginning of the test component behavior (Line 15). In Line 19, the timer is stopped before verdict setting. By means of the tracking class Timer4Lifeline, the relation between a test component and its timer can be made (Lines 1–1 and 21–23). By means of this information, the timer can be added to the test component classes as its attribute.

Figure 5.47 shows the excerpt of a test case diagram. In this example, a test case timer called Timer_CO is defined for the Test Coordinator lifeline, represented by the sand-clock symbol. After that, test behavior is defined. At the end of the test behavior, Timer Timer_CO is stopped and the test verdict of test component Test Coordinator is set to pass by means of the validation action. Thus, the test behavior of Test Coordinator for test case TestRoaming_noWarning is restricted to 100 seconds. If the time restriction is exceeded, a timeout of Timer_CO is catched in the default behavior of Test Coordinator.

Guarding Timer Timers may guard over test events and thus control the test behavior. Mostly, SUT responses are expected within a finite time duration after its triggering. So if the SUT does not response, the test fails. In order to detected such a failure in the system, a timer is started after a triggering event ist sent to the SUT with a certain expiration duration. It

Description: Add timer to a message of a certain lifeline in an Interaction Diagram. The Test Directives model specifies where a timer should be attached to and also to which instance the timer should be attached to.

```
 5 RULE AttachGuardingTimer2Lifeline(theTimer)
 6     FORALL ::uml2::Lifeline@UMLsrc theLifeline,
 7             :: uml2::Message@UMLsrc theMessage,
 8             :: testdirectives :: AddTimer@TestDirSrc theTimerAddition
 9     WHERE theTimerAddition.theLifeLine.name = theLifeline.name
10           // Take the lifeline
11     AND theTimerAddition.theLifeline.theMessage.name = theMessage.name
12           // Take the message on the lifeline where the timer should be attached
13     MAKE ::u2tp::timer theTimer
14           // Create a timer in the test case
15     SET theTimer.name = theTimerAddition.name,
16         // Set the name of the timer
17     IF theTimerAddition.opKind = Start
18       THEN SET
19             theLifeline . interaction .fragment.generalOrdering.after
20             .startExec.behavior.theMessage
21                           = theTimer.start(theTimerAddition.duration)
22             // Add start timer operation and attach timer to the lifeline
23             // after the message with the timer duration speficified
24             // in the testdirective model.
25     ELSEIF theTimerAddition.opKind = Stop
26       THEN SET
27             theLifeline . interaction .fragment.generalOrdering.after
28             .startExec.behavior.theMessage
29                           = theTimer.stop()
30             // Add stop timer operation and attach timer after message.
31     ENDIF
32     LINKING Timer4Lifeline
33         WITH theLifeline = theLifeline,
34              theTimer = theTimer
35 ;
```

Figure 5.48: Add Guarding Timer to Test Scenario

is stopped when the response comes within the predefined duration. Such a timer is called *guarding timer*.

However, for the specification of a guarding timer, a Test Directives model is required. By referencing to the event and the lifeline, the Test Directives model informs about the timer position in the diagram. Rule AttachGuardingTimer2Lifeline in Figure 5.48 shows the rule for guarding timer specification. First, the lifeline and the message for the timer are retrieved in Lines 9 and 11. After that, the timer is created and named accordingly as defined in the Test Directives model (Lines 13 and 15). If a timer start is specified in the Test Directives model, the duration of the timer must be

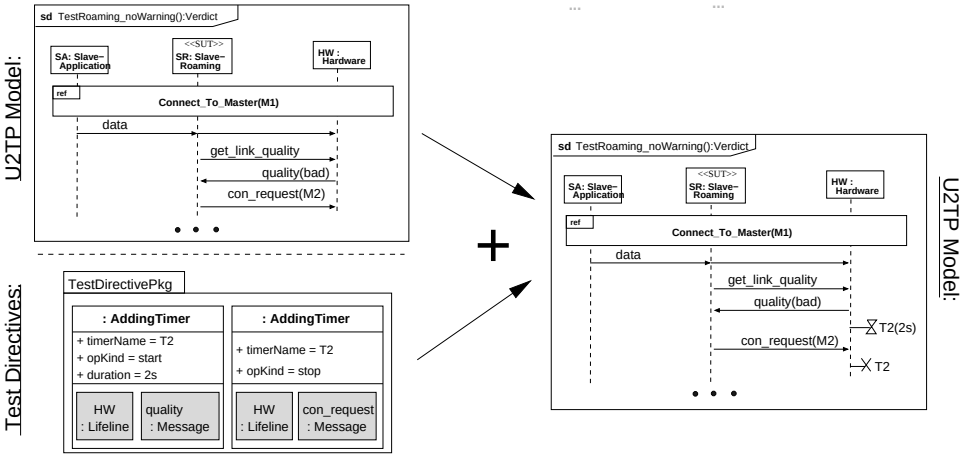


Figure 5.49: Specifying Guarding Timer

given. The timer is started or stopped after the given message (Lines 17–21 and Lines 25–29). Again, the guarding timer can be added to the test component classes as its attribute by means of the tracking class Timer4Lifeline (Lines 21–23).

Figure 5.49 visualizes the specification of a guarding timer in the Bluetooth test scenario. Since timer specification is a refinement step, a U2TP test diagram is the initial diagram for the specification of a guarding timer. The excerpt of test case TestRoaming_noWarning as shown in the left upper corner of the figure includes the SlaveApplication, SlaveRoaming as SUT and Hardware lifelines. The Test Directives model on the left lower corner specifies a start and a stop of a timer called T2 with an expiration duration of 2s. The timer is started after the SUT is triggered by the message quality on Lifeline HW and is stopped when SUT responses a con_request message within the expiration duration. The timeout of T2 is catched in the default behavior of Hardware.

5.4 Concluding Remarks

In this chapter, a guideline has been developed showing a way to derive U2TP test model from existing UML 2.0 system model. By means of well-defined transformation rules, we formalized the ideas introduced in the guideline. Furthermore, the Test Directives metamodel provides a good means to define test-specific requirements explicitely and to integrate them to the transformation. The transformation gets a UML 2.0 model and a Test Directives model as inputs. Its output is a skeleton of a U2TP model

whose granularity and preciseness strongly depends on those of the system model.

At time of doing this thesis, the UML 2.0 and MOF 2.0 standards were still working documents at the OMG. Thus, the transformation rules utilizing the UML 2.0 meta-classes are always adapted to the newest metamodel version. Also, for the realization of the test model transformation, tools enabling UML 2.0 and MOF 2.0 modeling are needed. Indeed, at time of the work, only a few exist. Furthermore, from the existing ones, only a subset of the UML 2.0 standard were implemented. For the specification and modeling of the Test Directives, we used an Eclipse plugin [Ecl06, Con06] combined with Rational Rose [IBM06b]. For the definition of the transformation rules, we found in Tefkat a reasonable QVT language.

As a last step in our MDT approach, test models shall be generated to executable test code. Herefore, the standardized test language TTCN-3 has proved itself a sophisticated executable test language.

Chapter 6

From Models to Test Code Generation

"Never send a human to do a machine's job." – *Morpheus, The Matrix*

In order to make UML 2.0 models "executable", the models need to be translated into executable code, such as Java or C++. This process is called code generation. Code generation from UML 2.0 models must be supported by tools mapping model concepts to target language constructs. U2TP as a domain-specific extension of UML 2.0 provides valuable means for functional and real-time test modeling. However, in order to make test models executable, they must be mapped to executable test languages as well.

In this chapter, we complete the test development process by generating executable test code from U2TP models. In doing so, we concentrate on test execution on system-level with TTCN-3. However, the development of TTCN-3 addresses only features for functional testing. Major concepts needed for real-time testing are missing.

In order to enable real-time integration and system tests, TTCN-3 is facilitated with real-time test concepts. This real-time extention introduces concepts for absolute time, synchronization of test components and evaluation of real-time requirements during and after a test run. It is called *Timed*TTCN-3. By means of mapping rules, *Timed*TTCN-3 code can be generated from a U2TP test model. To demonstrate the applicability of the mapping rules, we conclude the test development of the Bluetooth roaming approach by generating *Timed*TTCN-3 test code from its test model.

6.1 Motivation

Driven by the idea to make U2TP models executable, existing test languages and tools are examined by the U2TP consortium. However, U2TP has its roots in different test languages, among those are JUnit for unit testing and TTCN-3 for integration and system testing. For those languages, sophisticated tools already exist. Indeed, mapping U2TP models to the mentioned test languages enables the reuse of existing test infrastructures. As a result, a recommendation to map U2TP concepts to JUnit and TTCN-3 language constructs is supplied in the U2TP standard [U2T04]. Figure 6.1 illustrates the application areas of the mentioned languages: While U2TP can be utilized for test requirement, test design and test specifications, JUnit and TTCN-3 are mainly used to specify, implement, execute and evaluate tests during the test development process. However, TTCN-3 can also be used for test design[1]. Thus, Figure 6.1 shows overlapping areas between TTCN-3 and U2TP for test design and test specification phases. The application areas between JUnit and U2TP overlap in the test specification phase.

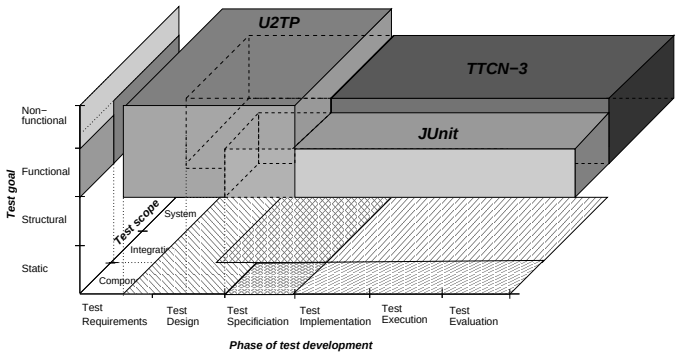


Figure 6.1: U2TP and TTCN-3 Application Areas

When looking at Figure 6.1, we note that while U2TP enables both functional and non-functional, i.e. real-time test modeling, TTCN-3 provides only limited means (i.e. timers and external functions) to enable real-time test specifications and JUnit does not support real-time test at all. As a consequence, U2TP models with real-time test properties cannot be executed.

Real-time systems testing is one of the most challenging research areas in testing. Such systems, especially distributed systems, are becoming even more important in daily life such as for business and administration (e.g. E-Commerce), home use (e.g. home brokerage), teaching (e.g. teleteaching

[1]The graphical representation of TTCN-3 called GFT[ETS05c] provides good means to design the test graphically.

and -tutoring) and process control (e.g. air traffic control) [WG99, RSG00, WSG98, Dep04]. For that, testing offers an important means to assure the correctness of distributed real-time systems with respect to functional and real-time behavior. Therefore, the focus of this work is laid on real-time test specification on system level. Real-time testing on unit level are topic of other works [JUn06b, Per03].

The procedures for testing functional system behavior are defined in the international ISO/IEC standard 9646 CTMF [ISO97]. As a part of CTMF, TTCN defines a language to specify functional system tests. Although TTCN-3 is not designed for real-time testing, it has still been used for specifying real-time tests [DST04]. Even its predecessor TTCN-2 has been successfully used to control external devices for performance testing [GKS00].

Indeed, using standard TTCN-3 for specifying real-time tests has some limitations although real-time properties may be accessed by timers and external functions. To be able to assess time properties within TTCN-3, its timer construct has to be used intensively. The drawback of using timers is that readability of test specifications suffers significantly. Furthermore, timers are always local to a test component. Thus, it is impossible to test real-time properties which are imposed on events occurring at different components [Neu04].

As a result, we conclude that timers can be used for detecting or provoking the absence of messages and to take care that a test case terminates properly, but it should better not be used for specifying real-time requirements. Also, accessing real-time properties by external functions is cumbersome, time-consuming and distorts the output. For distributed real-time testing, other concepts for dealing with time are preferable.

6.1.1 Related Work on TTCN with Real-Time Properties

Real-time extensions for TTCN as well as a formalised evaluation of log files are not completely new. In [UHPB03], an approach for a formalised analysis of log files is presented. Herein, log files of a distributed system are obtained by monitoring and later-on transformed into a formal SDL [ITU99] representation which can be checked by a model checker. Unfortunately, this approach involves several activities which cannot be specified and automatically executed as part of a test case. Furthermore, it does not address real-time properties but trustworthiness requirements.

However, two approaches were defined extending TTCN-2 for real-time and performance testing. They are called Performance TTCN (PerfTTCN) [SSR97] and Real-Time TTCN (RT-TTCN) [WG97, WG99].

PerfTTCN extends TTCN-2 with concepts for performance testing. These concepts are: (1) performance test scenarios for the description of test configurations, (2) traffic models for the description of discrete and continuous streams of data, (3) measurement points as special observation points, (4) measurement declarations for the definition of metrics to be observed at measurement points, (5) performance constraints to describe the performance conditions that shall be met, and (6) performance verdicts for the judgement of test results.

The PerfTTCN concepts are introduced mainly on a syntactical level by means of new TTCN tables. Their semantics are described in an informal manner. The traffic models may serve as input parameters for external load generator components and it is to be debated, if such information should be part of the compilable test suite or part of accompanying documents.

RT-TTCN is an extension of TTCN-2 for testing *hard* real-time requirements. On the syntactical level, RT-TTCN supports the annotation of TTCN-2 statements with two timestamps for earliest and latest execution time. On the semantical level, the TTCN-2 snapshot semantics has been refined and, in addition, RT-TTCN has been mapped onto timed transition systems [HMP91].

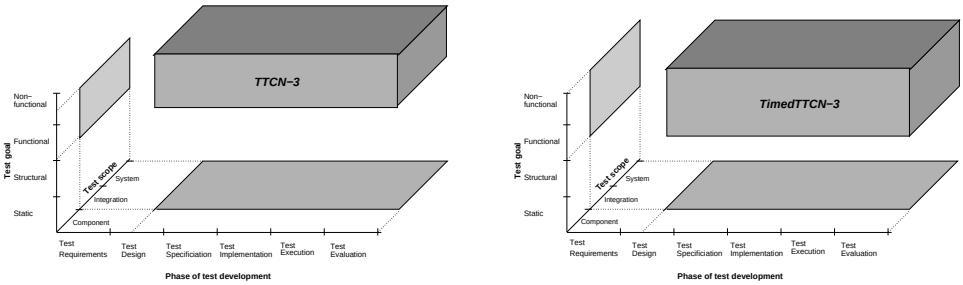
Although the RT-TTCN timestamps are introduced formally and look very simple, it turned out that their usage is not that simple. A timestamp associated with one statement is relative to the occurrence of previous events and define a time interval in which the statement is activated, i.e. can be executed. The handling of these activation intervals is not intuitive, especially if several statements can be executed, i.e. their activation intervals overlap. For the user it seems to be more natural to record execution times and to compare their values afterwards than to define activation intervals.

Both PerfTTCN and RT-TTCN are real-time extensions for TTCN-2. For TTCN-3, no real-time exist. Thus, a real-time extension for TTCN-3 and its relationship to U2TP models are needed. Here, a possible solution combining TTCN-3 with real-time concepts called *Timed*TTCN-3 is introduced.

6.2 A Real-Time Extension for TTCN-3

*Timed*TTCN-3 is a real-time extension to TTCN-3 that supports the test and measurement of real-time requirements [DGN02, DGN03]. It (1) introduces a *new test verdict* to judge real-time behavior, (2) supports *absolute time* as a means to measure time and to calculate durations, (3) allows to *delay* the execution of statements for defining time dependent test behavior, (4) supports the specification of *synchronization conditions* for test components and (5) provides a means for the *online and offline evaluation* of real-time properties.

Figure 6.2 depicts the application areas of traditional TTCN-3 and its real-time extension. While with the traditional TTCN-3, a tester has to specify real-time tests explicitely by timers and external functions, *TIMED*TTCN-3 supplies concepts to define real-time test properties in a more efficient way. Experiments give evidence that *TIMED*TTCN-3 covers most PerfTTCN and RT-TTCN features while being more intuitive in usage. Moreover, the *TIMED*TTCN-3 extensions are more unified than the other extensions by making full use of the expressiveness of TTCN-3. Thus only a few changes to the language are needed[2]. In the following, the real-time concepts of *TIMED*TTCN-3 are outlined.



a) Application Area of TTCN−3

b) Application Area of TimedTTCN−3

Figure 6.2: TTCN-3 and *TIMED*TTCN-3 Application Areas

6.2.1 Non-Functional Verdicts

The TTCN-3 verdicts indicate whether a test case is successful (**pass**), inconclusive (**inconc**) or faulty (**fail**) with respect to functional requirements. By introducing the possibility to test non-functional requirements, additional information concerning the test outcome is needed: A test case may pass with respect to both functional and non-functional behavior or it may pass only with respect to the functional behavior while the non-functional requirements are violated[3].

Since non-functional behavior can be observed only in combination with functional behavior on which the non-functional requirements are imposed, it is not meaningful to make any statements on non-functional test results if the functional behavior is not conforming to the functional requirements[4].

[2]Change requests for *TIMED*TTCN-3 and its graphical presentation are submitted to ETSI where TTCN-3 is developed and maintained [Neu02, Dai03].

[3]In the following, the terms *functional pass*, *non-functional pass*, etc. are used to describe the test outcome with respect to functional and non-functional behavior.

[4]In the special case of a tester malfunction which may lead to a wrong real-time measurement, the **error** verdict will be set by the tester.

Current value of verdict	New verdict assignment value				
	pass	conf	inconc	fail	none
pass	pass	conf	inconc	fail	pass
conf	conf	conf	inconc	fail	conf
inconc	inconc	inconc	inconc	fail	inconc
fail	fail	fail	fail	fail	fail
none	pass	conf	inconc	fail	none

Figure 6.3: *Timed*TTCN-3 overwriting rules for the test verdicts

Even in case of a *functional inconclusive*, no statement can be made on non-functional test results, since such an inconclusive case may have other non-functional requirements than the pass case which is the subject of testing. Hence, distinctive verdicts are just needed in case of a *functional pass*.

Besides the existing **pass** verdict which is used to indicate a *functional pass* with an associated *non-functional pass*, *Timed*TTCN-3 introduces the new verdict **conf** (as abbreviation for *conforming*) to indicate a *functional pass* with an associated *non-functional fail*. Due to the introduction of the new verdict, the overwriting rules for verdicts are refined. They are presented in Figure 6.3. The new verdict **conf** is inserted between the verdicts **pass** and **inconc**, with the overwriting rule: $pass < conf < inconc < fail < none$.

An example for the usage of the new **conf** verdict can be found in Figure 6.4. It shows an excerpt of an online evaluation function defined in the Bluetooth test. If the real-time requirement which is checked in the **else** statement in line 8 violates, **conf** is assigned as the local verdict of the appropriate test component (Line 9). Due to the overwriting rules of *Timed*TTCN-3, a **conf** verdict will not be overwritten by a **setverdict**(**pass**) statement, such as shown in Line 7. Also, a potential **fail** verdict is still available to express the functional fail.

```
1    if ((timeDifference > lowerbound) and (timeDifference < upperbound)){
2        return pass; //non−functional pass
3    } else {
4        return conf; //non−functional fail
5    }
```

Figure 6.4: Setting Real-Time Verdict

6.2.2 Time Concepts

For the handling of time, TTCN-3 provides the *timer* mechanism. This timer mechanism is designed for supervising the behavior of an IUT, e.g. to prevent the blocking of a test case or to provoke exceptional behavior. But it is not efficient enough to test and measure real-time properties, whereas the

latter is influenced by the TTCN-3 snapshot semantics and by the order in which the port queues and the timeout list are examined. TTCN-3 makes no assumptions about the duration for taking and evaluating a snapshot. Furthermore, the snapshot semantic has not been designed with having real-time testing in mind. Thus, exact time cannot be measured. An important prerequisite to solve the snapshot problem is to have a test system fast enough in a way that events for which realtime requirements apply are not queued but immediately processed. But this is out of the scope of this work. Instead, *TIMED*TTCN-3 provides language concepts to specify real-time test properties efficiently.

However, TTCN-3 also has no concept for *absolute time*, i.e. a test component cannot read and use its local system time. In real-time testing, the absolute time is necessary to check relationships between observed test events and to coordinate test activities. In case of synchronized clocks in a distributed test environment, the system time may be exchanged among test components to check real-time requirements that cannot be measured locally or for the timely coordination of test activities.

As a consequence of these considerations, *TIMED*TTCN-3 has the concept of *absolute time* in order to support real-time testing. In case of a distributed test environment, the test cases may define the requirements for the synchronization of clocks of different test components.

Absolute Time

Absolute time is related to *clocks* that provide the actual value of time. We assume that each test component has access to such a clock, but make no assumptions about the number and the synchronization of these clocks[5]. Furthermore, we assume that the resolution of such clocks and the speed of the tester are adequate for the real-time requirements which are subject of testing.

For the handling of time values either a new type is needed, or the time values have to be mapped onto an existing basic type. Due to numerous possible time representations, e.g. the UNIX approach to count the seconds since 1.1.1970 [IEE96] or a structured type with fields for year, month, day, hour etc., a common new type for time values is not easy to define. For simplicity, *TIMED*TTCN-3 uses the existing **float** type and follows the UNIX approach, i.e. time is counted in seconds and the absolute time is represented by the number of seconds since a fixed point in time. In contrast to the UNIX scheme, *TIMED*TTCN-3 does not define a fixed starting point for the time

[5]From a conceptional point of view, synchronized test components share the same clock, even though in a real implementation, the clocks are synchronized by using a synchronization protocol [Lam78, RSB90].

measurement. For that, *TIMED*TTCN-3 supports the usage of absolute time by the operations **now** and **resume**:

- The **now** operation is used for the retrieval of the current *local time*. The local character of the **now** operation is reflected by its application to the **self** handle, i.e. **self.now** is the expected call statement for the **now** operation. Operation **now** returns a float value that equals the current absolute time when the operation is called. The mapping of the float value onto a concrete daytime, e.g. year, month, day, hour, etc., is considered to be outside the scope of *TIMED*TTCN-3 and has to be provided by the test equipment, e.g. in form of additional conversion functions.

- The **resume** operation provides the ability to delay the execution of a test component. The argument of the **resume** operation is considered to be an absolute time value, i.e. the point in time when the test component shall resume its execution. If required, a relative time can easily be specified by using the current time as reference time, e.g. waiting for 3 seconds can be described by **resume**(**self.now** + 3.0).

```
1       for (var integer i:=1; i<=100; i:=i+1) {
2           resume(sendTime); // Send data packets every 0.5 time units.
3           saSUT.send(Data_config_Type_Template2);
4           log(TimestampType:{self.now, "SA!Data", i});
5           sendTime := self.now + 0.5;
6       }
```

Figure 6.5: Utilizing Absolute Time

An example for absolute time is shown in Figure 6.5. In this code fragment, data packages are sent hundred times and time is measured after each sending of a data package (Line 4). Furthermore, the data packages are sent periodically every 0.5 seconds by utilizing the **resume** operation in Lines 2 and 5.

Synchronization of Clocks

Time values are observed and used locally by the test components. Time values that are observed in different test components can be exchanged and used for further computations. But this only makes sense if the clocks of the involved test components are synchronized. The synchronization itself is outside the scope of *TIMED*TTCN-3 and should be guaranteed by the test equipment, but requirements for clock synchronization may very well be expressed in *TIMED*TTCN-3. These requirements can be used by a

*Timed*TTCN-3 compiler to distribute test components in an adequate manner or by a *Timed*TTCN-3 runtime environment to execute synchronization procedures for the test devices.

Timezones Most specification and implementation languages either support *local time* or *global time*. Local time means that each behavioral entity, such as a TTCN-3 test component has its own local time. Global time means that all behavioral entities share the same *global* time. Global time is perfect for the purpose of real-time testing, because all test components have by definition the same global time and are synchronized.

However, neither local nor global time are realistic assumptions for real-time testing situations. A real-time test environment typically consists of several devices. If synchronization among two or more test components is required to reach the goal of a test case, the components have to be executed either on the same device or on synchronized devices. The developer of real-time test cases should not care about synchronization procedures and the distribution of test components himself, but he can support their implementation by identifying test components which have to be synchronized. For this purpose, *Timed*TTCN-3 supports the *timezones* concept.

A timezone is an optional attribute that can be assigned to a test component when the component is created. Test components with the same attribute are considered to be synchronized, i.e. they have the same absolute time. A test component can only have one timezone attribute. Components without timezone attributes are considered to be not synchronized with other components. *Timed*TTCN-3 realizes the timezones concept by using an enumeration type with the reserved name **timezones**. The user has to specify the timezone attribute values by defining the **timezones** type in the module definitions part of a *Timed*TTCN-3 module. This type has an implicit member of name **none** which indicates no clock synchronization. The usage of an enumeration type only makes sense if the number of timezones is finite and known. We believe that this is a realistic assumption.

The timezone attribute of an **mtc** is assigned by using the **execute** operation. Attributes of all other test components are assigned by means of the **create** operations. The definition of a timezone type and the timezone assignment of the MTC by augmenting the **execute** statement are shown in Figure 6.6 and Figure 6.10. Figure 6.6 presents the definition of timezones Berlin, NewYork and Shanghai. Subsequently, the MTC is assigned to timezone Shanghai when executing test case TestCase_Timed in the control part (Lines 5–6 in Figure 6.10).

The flexibility of the timezones concept can be improved by making the timezones visible to the test components. This is realized in *TIMED*TTCN-3 by means of a special **timezone** function which returns the timezone of the component that called the function. In the case of a non-synchronized test component, the value **none** is returned. Like the **now** operation, the **timezone** operation is always applied to the **self** handle of a test component, i.e. **self.timezone** is the expected call statement for the **timezone** operation. The timezone information may be exchanged among test components to check if synchronization conditions are satisfied, or used to create several synchronized components.

```
1   type enumerated timezones {
2       Berlin , New York, Shanghai
3   }
```

Figure 6.6: Definition of Timezones

6.2.3 Evaluation of Real-Time Properties

While functional behavior is basically tested by using sequences of **send** and **receive** operations, real-time requirements can be tested by relating particular points in time to each other [ATM99b, ATM99a, BCS94]. The essence of the various real-time requirements can be broken down to the relationship of points in time. Mathematical formulae can be used to evaluate whether the points in time of interesting events fulfill a certain real-time requirement or not.

To obtain those points in time, functional TTCN-3 test cases are instrumented by statements which generate timestamps. *TIMED*TTCN-3 realizes this approach by making use of the possibility to read absolute time values which serve as timestamps. The mathematical formulae which are applied on the timestamps can be coded as TTCN-3 functions. Those *evaluation functions* return a judgement which indicates whether a requirement is fulfilled or not. An *online* or an *offline evaluation* of timestamps is possible:

- *Online evaluation* is needed if it is not possible to separate functional and non-functional requirements, i.e. a non-functional property directly influences the functional behavior of a testcase. In such a case, evaluation of non-functional observations must be performed during the test run in order to react on the result of the evaluation. Online evaluation has the drawback of cluttering the testcase and slowing down the performance of the testcase which may be undesirable for time-critical testcases.

- *Offline evaluation* may be used if the non-functional requirements which are subject of testing have no influence on the functional reaction of a testcase. In this case, the code just needs to be instrumented by statements that log the relevant timestamps. Based on the timestamps in the log file, the non-functional properties can be evaluated when the test run has finished. Offline evaluation has the advantage of having a low impact on the performance of a testcase, since only timestamps have to be logged during the test run.

Online Evaluation

For performing online evaluation, timestamps have to be evaluated during the test run, e.g. by calling a special evaluation function with timestamps as actual parameters. In a distributed test architecture, non-functional requirements may involve timestamps which have been collected by different test components. In this case, the evaluating component needs to obtain timestamps from other components. To achieve this, timestamps can be communicated directly among test components by using coordination messages. In order to evaluate tests online, the new concepts of *TIMED*TTCN-3 which have been introduced so far, are sufficient.

```
1   function onlineEval(float time1, float time2,
2                       float lowerbound, float upperbound)
3       return verdicttype
4   {
5       var float timeDifference := time2 − time1;
6       if ((timeDifference > lowerbound) and (timeDifference < upperbound)){
7           return pass; //non−functional pass
8       } else {
9           return conf; //non−functional fail
10      }
11  }
```

Figure 6.7: The Online Evaluation Function

The evaluation function onlineEval in Figure 6.7 checks the condition related to latency ($\mathsf{lowerbound} < t_{time2} - t_{time1} < \mathsf{upperbound}$). Depending on the result, it returns either a **pass** or a **conf** verdict (Lines 7 and 9) which may be used by the calling entity for further decisions.

Offline Evaluation

When using offline evaluation, the evaluation function is merely called after the test execution. *TIMED*TTCN-3 offers a means to record timestamps in a log file during a test run in order to evaluate them afterwards. The

final test verdict is a composition of the functional test verdict and the offline evaluation. To enable offline evaluation of real-time requirements, *TIMED*TTCN-3 refines the existing logging concept of TTCN-3.

The traditional TTCN-3 assumes that one global or several local log files exist and provides a **log** statement to enable logging of comments[6]. The number of log files is not specified, the logging mechanism is not described and neither module control nor test components can access the global or local log files. For an efficient offline evaluation, module control and test components need to have access to the log files and the contents of the log files have to be specified more formally.

The Log File Concept A *TIMED*TTCN-3 log file is basically a list of values of arbitrary TTCN-3 types. A log file is of type **logfile** and it is possible to handle log files by means of variables or to pass them into functions.

Each *TIMED*TTCN-3 test component has its own local log file. A local log file is initialized when the owning component is created. When test execution finishes, i.e. the main test component terminates, the local log files are merged into a global one. *TIMED*TTCN-3 does not specify the internal mechanisms that are needed for storing and maintaining log files[7], but defines functions for retrieving and accessing the entries of a log file (Table 6.1).

Logging of Events *TIMED*TTCN-3 refines the TTCN-3 **log** statement in order to write information into log files. In TTCN-3, the argument of the **log** statement is an arbitrary fixed string, in *TIMED*TTCN-3, the argument can be the value of any arbitrary valid type. For offline evaluation, a special structured type with a timestamp field may be specified. Figure 6.8 shows the specification of a timestamp type. Each timestamp is an entry in the logfile consisting of the logged execution time of the timed-critical event, the name of the appropriate event and its index. A corresponding offline evaluation function may only consider log file entries of the special type in order to judge the fulfillment of the real-time requirement. In Line 4 of Figure 6.5, the logging of an message is exemplified.

Log File Operations For retrieving entries of a log file, *TIMED*TTCN-3 offers means for sorting a log file by a certain field of the log file's

[6]This simple **log** statement is defined in the second version of the TTCN-3 standard release [ETS02] on which our *TIMED*TTCN-3 work is based. Meanwhile, an extended **log** statement exist in the third version of the standard, following our change requests defined in [Neu02, Dai03].

[7]The mechanisms for storing and maintaining log files are considered to be implementation specific and therefore outside the scope of *TIMED*TTCN-3.

```
1    type record TimestampType {
2         float logtime,
3         charstring event,
4         integer number
5    }
```

Figure 6.8: Defining Timestamp Type

entries. Since a log file may contain values of arbitrary types, sorting and retrieving is only possible for a certain type which has to be specified. According to the order which is imposed by sorting, the first, the next or the previous log file entry may be retrieved. For this purpose, *TIMED*TTCN-3 uses an internal cursor which points to an entry in the log file. This cursor can be moved and the value at the current cursor position may be retrieved.

The operation **first** serves two purposes: It selects the entries of the log file by their types and sorts them. In addition, it moves the cursor to the first matching entry in the log file. The first parameter of **first** specifies the field which is used as a sorting key. This is done using the TTCN-3 template notation: A "**?**" indicates the field which is used as sorting key, all other fields must be set to "**-**". The type of the template is used to select the type of entries which are regarded by the log file operations presented in Table 6.1. The second parameter can be used to further restrict the value of the entry, i.e. the internal cursor is moved to the first entry that matches the second parameter. The same matching mechanisms which are available for TTCN-3 **receive** statements apply.

Operation name	*Return type*	*Function*
first(*sortkey, value*)	**boolean**	Select and sort log file by *sortkey* and move to first matching entry in the log file
next(*value*)	**boolean**	Move to the next matching entry
previous(*value*)	**boolean**	Move to the previous matching entry
retrieve	type of *sortkey* used as parameter of **first**	Retrieve entry from current log file position
getlog	**logfile**	Get log file

Table 6.1: Overview of *TIMED*TTCN-3 *logfile* operations

```
1   function offlineEval(charstring receivedMessage,
2                        integer count, float upperBound) return verdicttype
3   {
4       var float timeSum := 0.0, timeDiff := 0.0, averageArrivalTime := 0.0;
5       var logfile timelog := getlog; // Get the logfile
6       if (timelog.first(TimestampType: {?, −, −},
7           TimestampType:{?, receivedMessage, ?}) == true) {
8           stampA := timelog.retrieve; // Get first logfile entry.
9           for (var integer i:=2; i<=count; i:=i+1) {
10            if (timelog.next(TimestampType: {?, −, −},
11                TimestampType:{?, receivedMessage, i}) == true) {
12              stampB := timelog.retrieve; //Get next logfile entry.
13              timeDiff := stampB.logtime − stampA.logtime;
14              timeSum := timeSum + timeDiff;
15              stampA := stampB;
16            }
17            else {
18                return fail;  // Wrong nuber of messages indicates
19                              // functional problem.
20                }
21          }
22          averageArrivalTime:=deltaSum/(n−1); // Calculated delta value.
23          if ((averageArrivalTime < upperbound)
24              and (lowerbound < averageArrivalTime)) {
25              return pass;   // non−functional pass
26          }
27          else {
28              return conf;   // non−functional fail
29          }
30      }
31      return fail;
32            // Wrong number of messages indecates functional problem
33  }
```

Figure 6.9: Offline Evaluation Function

The operations **next** and **previous** place the internal cursor to the next matching entry after or before the current cursor position. The order to which **next** and **previous** refer to is imposed by the sorting resulting from operation **first**. The parameter of **next** and **previous** is used in the same way as the second parameter of **first**. More complex search operations may be build from these basic search operations. The three operations **first**, **next** and **previous** return **true** when the matching entry is found in the log file, otherwise **false**. The value of the last matched entry, i.e. the value at the current cursor position, can be retrieved by the **retrieve** operation.

An offline evaluation function is exemplified in Figure 6.9. Function offlineEval implements the mathematical formula for the mean inter-arrival time based on the collected time stamps. The mathematical formula is:

$$meanInterArrivalTime := \sum_{i=2}^{n}(t_{Message}(i) - t_{Message}(i-1))/(n-1).$$

In the beginning, the logfile is retrieved in Line 5. In order to iterate through all captured time stamps of the time-critical message, the operations **first** (Lines 6–7) and **next** (Lines 10–11) are used. Since the **first** operation sorts the log file by the logtime field, the time stamp entries are matched in ascending order. The **retrieve** operation (Lines 8 and 12) yields the value of the last successfully matched entry, which is used to calculate the mean inter-arrival time and verifies that the mean inter-arrival time falls within the interval (lowerbound, upperbound). Based on the evaluation result, the function returns either **pass** or **conf** (Lines 25 and 28). In case that the retrieval of the **first** or **next** log entry fails, the log file contains less matching time stamps than expected. This is an indication for a non-conforming behavior of the SUT. Hence, evaluation is aborted with a **fail** verdict (Lines 18 and 31).

For the handling of global log files, *TIMED*TTCN-3 introduces a new operation called **getlog** and extends the existing **execute** statement. The **getlog** operation is used to retrieve the log file to serve the offline evaluation (Table 6.1). *TIMED*TTCN-3 also allows to apply the **getlog** function to **self** handles, i.e. a test component may access its own log file in order to perform a local offline evaluation after the collection of timestamps. Line 5 in Table 6.9 shows the use of the **getlog** operation in the evaluation function offlineEval.

In *TIMED*TTCN-3, a test run[8] comprehends testing of functional and real-time system properties, whereas the latter is optional. When executing a test case, the functional test is evaluated first. If this is successful, real-time properties are evaluated. The call of an offline evaluation function is achieved by an optional parameter of the **execute** operation. Its notation is done in square brackets "[ ]"[9].

The specification of a real-time test case within a TTCN-3 module control part is shown in Figure 6.10. In the beginning, the functional test case TestCase_Functional is executed. If it passes, a second real-time test case TestCase_Timed is performed, with timezone Shanghai assigned to MTC. In the same test run, the offline evaluation function evalOffline is performed Lines 5–6. For setting the final verdict, The final verdict contains the results of both the functional and real-time

[8] By test run, we mean the execution of one *TIMED*TTCN-3 test case.

[9] The *testrun handle* concept as introduced in [DGN02], which is a pointer returned by the **execute** statement and gives access to the test verdict and global testlog of a test run, has been simplified by the optional parameter of the **execute** operation evaluating the functional and real-time tests in the same test run.

```
1   control {
2       if (execute(TestCase_Functional()) == pass)
3           // Running test case with online real−time evaluation
4           {
5               execute(TestCase_Timed(), Shanghai)
6                                   [ evalOffline (myLog, "HW?Data", 100, 1.0)];
7               // Running test case with offline evaluation
8           }
9   }
```

Figure 6.10: Module Control Part with Real-Time Evaluation

test cases whereas the overwriting rules as defined in Section 6.2.1 apply.

6.3 Mapping Test Model to Test Code

To enable system test execution with existing TTCN-3 test infrastructure, the U2TP standard provides a recommendation to map U2TP test models to TTCN-3 code. In fact, almost all U2TP stereotypes have direct correspondence to TTCN-3 testing concepts. While U2TP incorporates all concepts defined in TTCN-3, there are U2TP concepts unknown by TTCN-3, e.g. arbiter, data pool, data partition, timezone, test log are concepts which cannot be directly mapped to TTCN-3. Other concepts can be mapped indirectly to (external) TTCN-3 functions implementing these concepts [U2T04].

In this work, we augment the mapping table provided in the U2TP document for *Timed*TTCN-3. Here, some U2TP concepts which cannot be mapped to TTCN-3, such as timezone and test log, are mapped to *Timed*TTCN-3. Consequently, real-time test code generation is enabled. In Tables 6.2, 6.3 and 6.4, mapping rules between U2TP and *Timed*TTCN-3 are listed. The mapping rules are just a recommendation like the one provided in the U2TP standard as mapping between two languages can be manifold. The principals for the mapping definition comprises two steps:

1. Assign U2TP stereotypes to *Timed*TTCN-3 concepts.
2. Implement U2TP concepts explicitely as *Timed*TTCN-3 functions if no direct mapping can be found.

Most of the mapping rules are quite straightforward or at least there are work-arounds. Concepts like SUT, test components, test cases, defaults, test control, verdicts, wildcards, test log, timer and timezone are provided in both languages. Thus, they can be mapped almost directly.

U2TP	*TIMED*TTCN-3
Test Architecture:	
System Under Test (SUT)	Accesses SUT via abstract test system interfaces with SUT interfaces result in port types. An additional port is needed to communicate with a user-defined arbiter.
Test component	*TIMED*TTCN-3 component type used for creation of test components and their connections to SUT and other test components.
Test context	*TIMED*TTCN-3 module covering all test cases, a specific behavioral function to set up the initial test configuration for this test context, the test control and other test relevant definitions.
Test configuration	Specific behavior function to set up the initial test configuration. This function utilizes *TIMED*TTCN-3 operations to create, start, connect/disconnect, map/unmap the dynamic test configuration.
Arbiter	The U2TP default arbiter is a *TIMED*TTCN-3 builtin. User-defined arbiter can be realized by the MTC or an additional component in the test system collecting local verdicts from test components and evaluate them according to arbitration definition after the test run.

Table 6.2: Mapping Rules for Test Architecture

Nevertheless, *TIMED*TTCN-3 and U2TP differ in some syntactic as well as semantic aspects: While U2TP is based on object oriented paradigm of UML 2.0 where behavior is bound to objects, *TIMED*TTCN-3 is based on concept of functions and the binding of functions to test components [SDGR03]. Also, while *TIMED*TTCN-3 uses a FIFO queue per test component port, U2TP defines a semantic variation point about its FIFO queues for a test component[10]. Furthermore, while *TIMED*TTCN-3 supports dynamic configurations in terms of type and number of test components and their connection to the SUT and among each other, U2TP uses static configuration structures with predefined maximum number of test components. Additionally, U2TP may use different diagrams to model the same thing, e.g. a test case can be defined in Interaction Diagrams but also in State Machines or any other behavior diagram. In the following, we will outline the major mapping rules in Tables 6.2, 6.3 and 6.4.

In order to represent the SUT, *TIMED*TTCN-3 provides the indirect definition via ports from the test components to the SUT (Table 6.2). Its access

[10]In U2TP, a test component may have one FIFO or several FIFO queues.

by the test system is achieved via test system interfaces. As mentioned, test context is a very strong concept in U2TP and embodies diverse test artifacts like test control and test configuration. Thus, this concept is mapped to a *TimedTTCN-3* module as the top-level element in a test suite. The module embodies the participating test cases in the test run, behavior functions of the test components, a function to initialize the test configuration and the test control.

In general, *TimedTTCN-3* provides a built-in arbiter with predefined arbitration rules. In case that an user-defined arbiter is needed, it can be realized either directly by the MTC or by creating a new component in the *TimedTTCN-3* test system at the beginning of a test-run. In order to overrule the verdict mechanism of *TimedTTCN-3*, special verdict types are needed. Validation action are realized by external functions.

In U2TP, a test case can be modeled by any kind of behavior diagram (Table 6.3). Behavior diagrams can be distinguished in two categories providing either a local or a global view of an object. Diagram types providing a global view are Interaction and Activity Diagrams, where the behavior of all ob-

U2TP	*TimedTTCN-3*
Test Behavior:	
Test objective	A comment in a *TimedTTCN-3* test behavior.
Test case	The behavior of a *TimedTTCN-3* test case is defined by test case definition for the MTC and behavioral functions for PTCs. The MTC creates, controls and triggers the test components and the arbiter.
Defaults	*TimedTTCN-3* **altstep** definition with dynamic activation and deactivation of the default behavior.
Test control	Module control part of *TimedTTCN-3*.
Test invocation	Test case execution.
Stimulus, observation, coordination	Various *TimedTTCN-3* communication operation.
Verdict	*TimedTTCN-3* introduces a built-in verdict handling with standard overwriting rules for the verdict values. For user-defined verdicts, a special verdict type is needed.
Validation Action	External function or data functions resulting in a value of the specific verdict type.
Log action	Log operation.
Test log	Log file.

Table 6.3: Mapping Rules for Test Behavior

jects within a system is shown. State Machines concentrates on the behavior of a single object and thus provide a local view of this object. Test behavior defined in *TimedTTCN-3* only specify the local view of a test component. The behavior of MTC is specified in a *TimedTTCN-3* test case while the test behavior of the PTCs are defined in *TimedTTCN-3* behavior functions. They are started by MTC and run concurrently. The MTC also controls the PTCs and the arbiter in the test system.

Another difference between U2TP and *TimedTTCN-3* is the default handling. In contrast to *TimedTTCN-3* which uses function-based defaults with dynamic activation and deactivation during test execution, U2TP uses structural defaults bound to the structure of a test system. The latter impacts on a hierarchy of defaults, from test component level down to event level, and results in less dynamic default handling. Still, mapping U2TP default diagrams to *TimedTTCN-3* default functions can be achieved directly.

Verdicts defined in U2TP can also be mapped directly to functional verdict values in *TimedTTCN-3*. However, verdict handling in *TimedTTCN-3* is defined for conformance and real-time testing. In fact, U2TP does not provide real-time test verdicts. Instead, the pass verdict can be used for both functional and real-time tests. In order to distinguish between conformance and real-time test verdicts, additional verdicts and real-time arbitration functions must be defined by the user.

Stimuli, observation and coordination in a U2TP model exchange information between SUT and the test system. These message exchanges can be easily converted to *TimedTTCN-3* communication operations. *TimedTTCN-3* distinguishes between message-based communication and procedure-based communication for communication between SUT and test system and coordination within the test system.

The test log concept is not part of TTCN-3. But in *TimedTTCN-3*, it is introduced as a real-time test concept. However, *TimedTTCN-3* enhances the appliance of a log file by additional operations to sort and retrieve log file entries. This feature is not facilitated by U2TP. There, the log file has to be evaluated manually by the tester.

In a U2TP model, only UML 2.0 data are supported (Table 6.4). They embody primitive types (like boolean, string or integer) and classes, while *TimedTTCN-3* supports all types such as basic types (e.g. integer, char, verdicttype etc.), basic string types (e.g. bitstring, charstring etc.), user-defined structured types (record, set, enumerated etc.) and anytype. A U2TP data pool can be realized in a supplementary *TimedTTCN-3* module which incorporates the test data and is referenced in the test module. Also, *TimedTTCN-3* does not support concepts to partition and choose test data. In some way, *TimedTTCN-3* may use data matching mechanisms for template definitions to gain a set of test data. But since templates with matching mechanisms can only be defined for a single data type, these data belong to

U2TP	*TIMED*TTCN-3
Test Data:	
Wildcards	*TIMED*TTCN-3 data matching mechanisms.
Data pool	External constants referring to the data pool or functions accessing data pool.
Data partition	*TIMED*TTCN-3 matching mechanisms to handle data partitions for observations. For stimuli however, user-defined functions are needed to realize the test case execution with different data to be sent to tehdata pool SUT.
Data selector	An external function to get access to the data of a data pool or data partition.
Coding rules	*TIMED*TTCN-3 encode attributes.
Time:	
Timer	Timer and timer operations.
Timezone	*TIMED*TTCN-3 timezone.

Table 6.4: Mapping Rules for Test Data and Time

the same data type. Even using the grouping mechanism to arrange these data templates is not adequate to the data partition and data selection concepts in U2TP. One solution is to define external *TIMED*TTCN-3 functions to select data values to form data partitions. Furthermore, *TIMED*TTCN-3 allows data matching used only for observation. For stimuli, concrete test data are required. Thus, user-defined functions are needed to send different data to the SUT during test case execution while matching mechanisms can be utilized to handle data partitions for observations.

UML 2.0 enables the retrieval of absolute time and U2TP provides the timezone concept. Thus, these two concepts can be mapped directly between U2TP and *TIMED*TTCN-3 to evaluate real-time properties of a distributed test system. Accordingly, the timer concept and its operations can also be found in both languages.

Last but not least, even if mapping between U2TP and *TIMED*TTCN-3 allow direct test code generation, U2TP models and *TIMED*TTCN-3 code may show different levels of abstractions and thus may complicate the code generation or leads to incomplete code for test execution: while *TIMED*TTCN-3 is on a detailed test specification level where tests can be executed directly, U2TP can also be used on more abstract levels by defining just the principal constituents of, e.g. a test purpose or test case, without giving details needed for test execution. While the latter provides great advantage for the test design, additional means have to be taken to generate executable tests.

For example, the expressiveness of UML 2.0 Interaction Diagrams allows to describe a whole set of test cases by just one diagram, so that test generation methods have to be applied in order to derive these test cases from the diagram.

6.4 Executable Test Code for the Bluetooth Roaming Application

In this section, a *Timed*TTCN-3 test suite for the Bluetooth roaming application is generated from the U2TP model following the mapping rules[11]. We assume that the U2TP model is detailed enough to get adequate test code for its execution. The test code meets the test requirements given in Section 4.4.

6.4.1 Defining Test Architecture

A *Timed*TTCN-3 test architecture defines the interaction between the SUT and the test system by means of both the abstract and real test system interfaces. Communication between SUT and the test system is realized by mapping test component ports to test system ports and exchange messages via interfaces to the SUT. Queues in the ports are used to catch and store incoming messages. The test system owns a test system interface to the SUT. For test execution, the ports of the test components must be associated with the ports of the test system interface.

Figure 6.11 shows the distributed test architecture that is used by the *Timed*TTCN-3 test suite in order to test the roaming functionalities of the Bluetooth slave device. This architecture consists of the MTC that emulates the Test Coordinator and PTCs that emulate Slave Application, Master and Hardware[12]. In the presented test architecture, Hardware is emulated by software. In case that real Bluetooth hardware devices are used in the test, the SUT comprehends the real Bluetooth hardware devices with BTRoaming as Implementation Under Test (IUT). The test system emulates Test Coordinator, Slave Application and Master only.

6.4.2 Specifying Test Suite

In order to meet the *Timed*TTCN-3 test architecture described before, the U2TP test configuration has to be mapped to *Timed*TTCN-3 test configuration. As recommended in the mapping table (Table 6.2 on Page 141),

[11] In this section, only code snippets are shown. The complete test suite can be found in Appendix C.

[12] For a better illustration, Master and Hardware are shown in the same block in Figure 6.11. But for the real test implementation, they are separated PTCs.

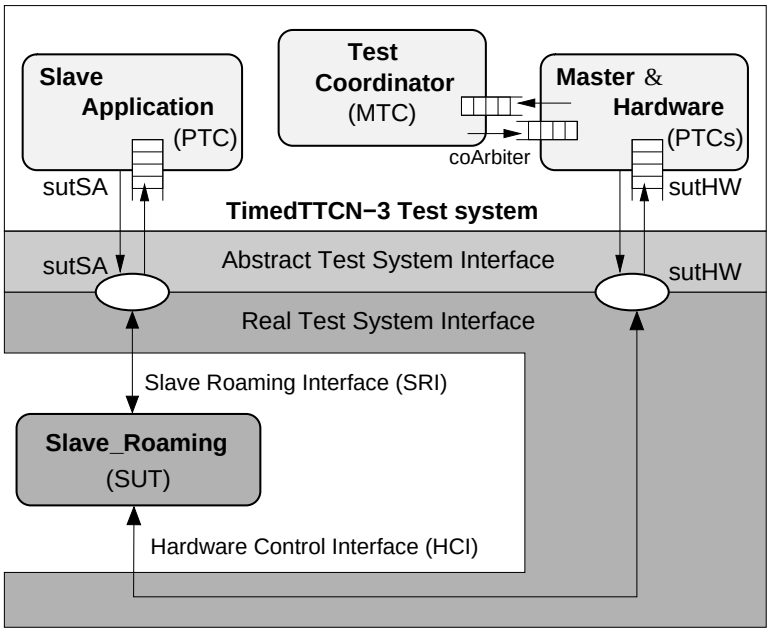


Figure 6.11: Test Architecture

test component types, their ports and the messages exchanged via the ports need to be specified. Herein, U2TP test components are mapped to *Timed*TTCN-3 component types. Ports in the test model are transformed to port types. Messages are generated to data templates.

Figure 6.12 shows the definition of data type Connection_Signalling_Type. It is a record type with two elements, indicating the message kind and the Bluetooth device address (Lines 1–6). Message kind represents either a connection request, a connection acception or a connection confirmation. In Lines 7–10, an appropriate template of data type Connection_Signalling_Type defines a connection request to be sent to master M1.

In contrast to U2TP, *Timed*TTCN-3 does not specify an SUT component explicitely. Instead, the test system accesses the SUT via abstract Test System Interfaces. Thus, the mapping table recommends to define port types for SUT interfaces used by the Test System Interfaces. To achieve this, a message-based port called Hardware_PType is defined to enable communication between the hardware and the SUT. Herein, messages of type Connection_Signalling_Type can be exchanged (Line 12). The Hardware_PType is utilized in the definition of SlaveRoaming_CType component type. By means of port Hardware_PType the SUT can be accessed through the *Timed*TTCN-3 Test System Interface (Line 16).

```
1   type record Connection_Signalling_Type {
2       charstring SigMessage,
3           // Defining a connection request,
4           // connection acception or connection confirmation.
5       Master_CType Master
6   }

7   template Connection_Signalling_Type conRequest_M1_Template := {
8       SigMessage :="con_request",
9       Master := M1
10  }

11  type port Hardware_PType message {
12      inout Data_Type, Connection_Signalling_Type,
13      RoamingList_Type, Link_Quality_Type;
14  }

15   type component SlaveRoaming_CType {
16       port Hardware_PType sutHW;        // port to Hardware
17       port SlaveApplication_PType sutSA; // port to Slave Application
18   }
```

Figure 6.12: Bluetooth Type Definitions

Guided by the mapping table, *Timed*TTCN-3 configuration operations enable dynamic creation and destruction of PTCs by the MTC. Moreover, test components are connected to each other and mapped to the SUT by connecting and mapping the ports. Figure 6.13 shows a function to establish the test configuration for the *Timed*TTCN-3 test suite. In this configuration, several PTCs are created (Lines 10–16). Following the mapping definition, message exchanges between the test system and the SUT is enabled by indirect definition of ports and mapping the SUT ports to the test component ports. Thus, in Lines 18–19, test components are mapped to SUT ports. Furthermore, test components are connected to each other (Lines 20–35).

Since the U2TP model in our case study makes use of an arbiter, it needs to be mapped to *Timed*TTCN-3 code. Following the mapping table in Section 6.3, special verdict types and arbiter specific artefacts like an arbiter port and component type need to be defined. Also, behavior of an arbiter must be specified explicitely.

Figure 6.14 implements a user-defined verdict type called TestVerdict_Type, comprehending functional and real-time verdicts pass_, conf_, inconc_, fail_, error_ and none_. The "_" is mandatory in order to distinguish between the existing *Timed*TTCN-3 keywords and the user-defined arbiter values. Furthermore, an arbiter component type Arbiter_CType and its utilized port type Arbiter_PType are specified. Since an arbiter is a component in the

```
1   function System_Config(inout Arbiter_CType Arbiter,
2           inout SlaveApplication_CType SA, inout SlaveRoaming_CType sut,
3           inout Hardware_CType HW, inout Coordinator_CType coordi,
4           inout Master_CType M1, inout Master_CType M2,
5           inout Master_CType M3, inout Master_CType M4)
6       runs on Coordinator_CType
7   {
8       var MasterList_Type MasterList;
9       //    creation of PTCs:
10      SA := SlaveApplication_CType.create;
11      HW := Hardware_CType.create;
12      M1 := Master_CType.create;
13      M2 := Master_CType.create;
14      M3 := Master_CType.create;
15      M4 := Master_CType.create;
16      Arbiter := Arbiter_CType.create; // Create the arbiter .
17      // Mapping & connecting ports:
18      map(sut: sutSA, SA: saSUT);
19      map(sut: sutHW, HW: hwiSUT);
20      connect(HW: hwiMaster, M1: masterHW);
21      connect(HW: hwiMaster, M2: masterHW);
22      connect(HW: hwiMaster, M3: masterHW);
23      connect(HW: hwiMaster, M4: masterHW);
24      connect(HW: hwiArbiter, Arbiter: ArbiterPort);
25      connect(coordi: coMaster, M1: masterCo);
26      connect(coordi: coMaster, M2: masterCo);
27      connect(coordi: coMaster, M3: masterCo);
28      connect(coordi: coMaster, M4: masterCo);
29      connect(coordi: coArbiter, Arbiter: ArbiterPort);
30      // Connecting arbiter with other test components:
31      connect(SA: saArbiter, Arbiter: ArbiterPort);
32      connect(M1: masterArbiter, Arbiter: ArbiterPort);
33      connect(M2: masterArbiter, Arbiter: ArbiterPort);
34      connect(M3: masterArbiter, Arbiter: ArbiterPort);
35      connect(M4: masterArbiter, Arbiter: ArbiterPort);
36  }
```

Figure 6.13: Creating Test Configuration

test system, it is to be created by the MTC during the test configuration establishment and connected to other test components in the test system. This is realized in the test configuration function in Lines 16 and 31–35 of Figure 6.13.

According to the mapping recommendation, arbiter behavior must be defined explicitely in a *TIMED*TTCN-3 function. Figure 6.15 shows the arbitration function specified in the test suite. In the beginning, the verdict variables are initiated by the neutral verdict `none_`. During the test run, the arbiter collects verdicts from the test components involved in the test (Lines 11–25). After the test is finished, the collected verdicts are evaluated

```
1   type enumerated TestVerdict_Type {
2       conf_,  // conf_ is a real-time verdict.
3       pass_,  // functional verdicts
4       inconc_,
5       fail_,
6       error_,
7       none_,
8   }

9   type port Arbiter_PType message {
10      inout TestVerdict_Type
11  }

12  type component Arbiter_CType {
13      port Arbiter_PType ArbiterPort;
14  }
```

Figure 6.14: Arbitration Types

according to the arbitration rules. According to the overwriting rules for the *TIMED*TTCN-3 test verdicts, the evaluation order is error > fail > inconc > conf > pass > none. This is implemented in Lines 26–40.

In order to describe the objective of a test and provide its concrete specification, *TIMED*TTCN-3 test cases are needed. Following the mapping rules in Table 6.3 on Page 142, test objective description is not part of *TIMED*TTCN-3, but can be realized by comments in a test case specification. Figure 6.16 shows the specification of the functional test case called TestRoaming_noWarning derived from the U2TP test case with the same name (cp. Figure 4.17 on Page 74). Its objective can be found as comments in Lines 1–3. In Lines 7–16, variable declarations are made and the test system is setup by the configuration function System_Config described before (Lines 17–18). Furthermore, the concurrent behavior of the PTCs are started by the MTC (Lines 21–26). The emulation of the Bluetooth link quality is specified in the behavior of the Hardware in Line 22[13].

Following the mapping table (Table 6.4 on Page 144), U2TP timers and timer operations are straightly mapped to *TIMED*TTCN-3 timers and timer operations. Thus, the U2TP test scenario defining a test case timer T1 of six seconds is mapped to a *TIMED*TTCN-3 timer with the same name and the same expiration duration. This timer is started in Line 28 and stopped at the end of the test case behavior in Line 46. A second timer called T2 is utilized to guard the connection request message is specified in this test case as well (Lines 31–33). Afterwards, Slave is connected to Master M1 (Line 29).

[13]The specification of Hardware_Behavior can be found in Appendix C.

```
1  function Arbitration(SlaveApplication_CType SA, Hardware_CType HW,
2                       Master_CType M1, Master_CType M2,
3                       Master_CType M3, Master_CType M4)
4      runs on Arbiter_CType
5  {
6      var TestVerdict_Type testVerdict1 := none_;
7      var TestVerdict_Type testVerdict2 := none_;
8      var TestVerdict_Type testVerdict3 := none_;
9      var TestVerdict_Type testVerdict4 := none_;
10     var TestVerdict_Type testVerdict5 := none_; // Initialize verdicts.
11     while (SA.running or HW.running or M1.running or
12            M2.running or M3.running) {
13         alt {   // Receiving verdicts from test components.
14             [] ArbiterPort.receive(TestVerdict_Type: pass_) {
15               testVerdict1 := pass_; }
16             [] ArbiterPort.receive(TestVerdict_Type: conf_) {
17               testVerdict2 := conf_; }
18             [] ArbiterPort.receive(TestVerdict_Type: inconc_) {
19               testVerdict3 := inconc_; }
20             [] ArbiterPort.receive(TestVerdict_Type: fail_) {
21               testVerdict4 := fail_ ; }
22             [] ArbiterPort.receive(TestVerdict_Type: error_) {
23               testVerdict5 := error_ ; }
24         }
25     }
26     if (testVerdict5 == error_) { // Definition of the arbitration rules
27         setverdict(error);
28     } else
29         if (testVerdict4 == fail_) {
30             setverdict(fail);
31         } else
32             if (testVerdict3 == inconc_) {
33                 setverdict(inconc);
34             } else
35                 if (testVerdict2 == conf_) {
36                     setverdict(conf);
37                 } else
38                     if (testVerdict1 == pass_) {
39                         setverdict(pass);
40                     }
41 }
```

Figure 6.15: Arbitration Function Definition

In the mapping table, U2TP message exchanges like stimulus, observation or coordination messages are recommended to be mapped to various *TimedTTCN-3* communication operation incorporating message-based and procedure-based communication operations. Since the U2TP test case in our test model specifies message-based exchanges, they are mapped to *TimedTTCN-3* **send** and **receive** operations (e.g. Lines 30 or 42). Refer-

```
1   // Functional test case TestRoaming_noWarning: Connection establishment
2   // between slave and master M1. After the exchange of two data
3   // packages, link quality becomes bad. If M2 cannot be reached, M3 is chosen.
4    testcase TestRoaming_noWarning()
5       runs on Coordinator_CType system SlaveRoaming_CType
6   {
7       var SlaveApplication_CType SA; var Hardware_CType HW;
8                 // Declaring test components
9       var Master_CType M1; var Master_CType M2;
10      var Master_CType M3; var Master_CType M4;
11      var Arbiter_CType Arbiter; // Declaring Arbiter component.
12      var RoamingList_Type roamingList;
13      var TestVerdict_Type verdict; // Declaring verdicts .
14      var default GeneralDefault; // Declaring defaults .
15      var default ConfDefault;
16      var timer T1, T2; // Declaring timers.
17      System_Config(Arbiter, SA, system, HW, self, // Test configuration is
18                    M1, M2, M3, M4); // built in the System_Config function.
19      MasterList := {M1, M2, M3, M4};
20              // Build the list with all masters within the network.
21      SA.start(SA_Behavior()); // Start test component behavior.
22      HW.start(Hardware_Behavior());
23      M1.start(Master1_Behavior());
24      M2.start(Master2_Behavior());
25      M3.start(Master3_Behavior());
26      Arbiter.start(Arbitration(SA, HW, M1, M2, M3)); // Start arbiter behavior.
27      GeneralDefault := activate(General_Default); // Activate test case default
28      T1.start(6.0);  // Start testcase timer T1 of 6 seconds.
29      Connect_To_Master_Coordi(M1, MasterList); // Establish connection to M1.
30      coMaster.receive(conRequest_M2_Template) from M2;
31      T2.start(2.0);  //Activation of guarding timer T2 for 2 seconds.
32      coMaster.receive(conRequest_M3_Template) from M3;
33      T2.stop; // T2 is stopped after receiving a connection request from M3.
34      T2.start(2.0); //Restart T2.
35      ConfDefault := activate(con_confirm_Default());
36                  // Activate message-based default for connection confirmation.
37      coMaster.receive(conConfirm_M3_Template) from M3;
38                  // Connection confirmation from the new master.
39      deactivate(ConfDefault); // Deactivate the message-based default.
40      T2.stop; // T2 is stopped after receiving a connection confirm from M3.
41      roamingList := MakeList(system, M1, M2, M4);
42      coMaster.send(roamingList) to M3;
43              // Test Coordinator sends new Roaming List to Slave.
44      Disconnect_Config(Arbiter, SA, system, HW, self,
45                        M1, M2, M3, M4);
46      T1.stop; // Stop the testcase timer.
47      deactivate(GeneralDefault);
48      verdict := pass_;  // Set the verdict
49      coArbiter.send(verdict);   //Send verdict to Arbiter
50  }
```

Figure 6.16: Test Case TestRoaming_noWarning

ences defined in the U2TP scenario are realized by *TIMED*TTCN-3 functions (Line 44–45). At the end of the test behavior, the test verdict is set. According to the mapping table, this can be achieved by setting a *TIMED*TTCN-3 verdict variable to pass_ and send the value explictely to the arbiter component. The setting of verdicts is realized in Lines 48–49 of the *TIMED*TTCN-3 test suite.

```
1   altstep General_Default() runs on Coordinator_CType {
2       [] coMaster.receive(*) { // Receive ANY message
3           coArbiter.send(TestVerdict_Type: inconc_);
4                   // Verdict set to inconclusive
5       }
6   }

7    altstep con_confirm_Default() runs on Coordinator_CType {
8       [] coMaster.receive(Connection_Signalling_Type: ?) {
9           // Receive wrong message parameter.
10          coArbiter.send(TestVerdict_Type: fail_);
11          // Verdict set to fail
12          stop; // Terminate.
13      }
14  }
```

Figure 6.17: General and Message-Based Defaults

In U2TP, default behavior is modeled separately from its activating test behavior. The mapping table recommend to map U2TP defaults to **altsteps** in *TIMED*TTCN-3 and activate them in the test case. For that, wildcards utilized in the U2TP default specification are transformed to *TIMED*TTCN-3 data matching mechanisms which also contain wildcard definitions.

In Figure 6.17, two defaults are specified. The first default called GeneralDefault catches all unexpected messages sent to the Test Coordinator and returns the inconclusive verdict to the arbiter component. It is activated at the beginning of the test behavior and deactivated at the end of the test case (Lines 27 and 47 in Figure 6.16). Furthermore, a second default ConfDefault is activated to handle connection confirmation messages with unexpected parameter values from Master M3. This default is activated before the reception of message con_confirm and deactivated after the reception of the message with correct parameters (Lines 35 and 39). In case of invocation of the default behavior, the test verdict concludes **fail_**.

In the U2TP model, a second test case TestRoaming_DataSending is specified to send one hundred data packages to the SUT (cp. Figure 4.21 on Page 78). In this test case, time is logged and measured for each data package sent from Slave Application to SUT and subsequently forwarded to Hardware. There, the average tolerance for sending two data packages, i.e. jitter, should be evaluated. According to the mapping recommendation, real-time properties

```
1   type record TimestampType {
2       float logtime,
3       charstring event,
4       integer number
5   }

6   type enumerated timezones {
7       Berlin, NewYork, Shanghai
8   }
```

Figure 6.18: TimestampType and Timezone Definitions

are mapped to newly defined *TIMED*TTCN-3 real-time concepts. For that, the U2TP log action is mapped to the *TIMED*TTCN-3 log operation and a U2TP test log is mapped to a *TIMED*TTCN-3 log file. In order to generate *TIMED*TTCN-3 code for the given U2TP test scenario, *TIMED*TTCN-3 data types are first needed to specify timestamps and timezones. The logging of *TIMED*TTCN-3 timestamps and storing those to a log file correspond to the logging of time values in a U2TP test log by log actions. Figure 6.18 shows the type definitions for timestamp and timezone. This TimestampType incorporates the time value, the message name and the index number of the data package. Additionally, the test components defined in the *TIMED*TTCN-3 test system may belong to the timezones Berlin, NewYork or Shanghai.

Figure 6.19 shows the real-time test behavior of the hardware test component which is started by the MTC in test case TestRoaming_DataSending. The

```
1   function Hardware_Behavior_DataSending()
2       runs on Hardware_CType
3   {
4       var Master_CType theMaster;
5       var TestVerdict_Type verdict;
6       Connect_To_Master_Hardware(Master);
7       for (var integer i:=1; i<=100; i:=i+1) {
8           hwiSUT.receive(Data_Type_Template1);
9           // Logging of data reception:
10          log(TimeStamp:{self.now, "HW?Data", i});
11          theMaster := Data_Type_Template1.Master;
12          hwiMaster.send(Data_Type_Template1) to theMaster;
13          // Trigger good link quality:
14          Link_Quality(good);
15      }
16      verdict := pass_;
17      coArbiter.send(verdict);
18  }
```

Figure 6.19: Hardware Test Component with Real-Time Properties

```
1   function evalJitter(charstring receivedMessage,
2           integer count, float upperBound) return TestVerdict_Type
3   {
4       var float deltaSum := 0.0, deltaDiff := 0.0, delta := 0.0;
5       var float jitterSum := 0.0, jitterDiff := 0.0, jitter := 0.0;
6       var logfile timelog := getlog;
7                                   // Get the logfile
8       //Calculate delta for the mean inter arrival time:
9       if (timelog.first(TimestampType: {?, −, −},
10              TimestampType:{?, receivedMessage, ?}) == true) {
11          stampA := timelog.retrieve; // Get first logfile entry.
12          for (var integer i:=2; i<=count; i:=i+1) {
13            if (timelog.next(TimestampType: {?, −, −},
14                  TimestampType:{?, receivedMessage, i}) == true) {
15              stampB := timelog.retrieve; //Get next logfile entry.
16              deltaDiff := stampB.logtime − stampA.logtime;
17              deltaSum := deltaSum + deltaDiff;
18              stampA := stampB;
19            }
20          }
21          delta:=deltaSum/(count−1); // Calculated delta value.
22        // Calculate jitter :
23        if (timelog.first(TimestampType: {?,−,−},
24             TimestampType:{?, receivedMessage, ?}) == true) {
25          stampA := timelog.retrieve; // Get first logfile entry.
26          for (var integer i:=2; i<=count; i:=i+1) {
27            if (timelog.next(TimestampType: {?, −, −},
28                TimestampType:{?, receivedMessage, i}) == true) {
29              stampB := timelog.retrieve; //Get next logfile entry.
30              deltaDiff:=stampB−stampA;
31              jitterDiff := delta−deltaDiff;
32                if ( jitterDiff < 0) {
33                  jitterDiff := (−1) ∗ jitterDiff ;} // Absolute value
34                if ( jitterDiff > upperBound) { // single jitter too big
35                    return conf_;}
36                else {
37                    return pass_; }
38              stampA:=stampB;
39            }
40            return pass_;
41          }
42        }
43      }
44  }
```

Figure 6.20: Offline Evaluation Function for Jitter

*Timed*TTCN-3 code is generated from the U2TP test case diagram shown on Page 78. In Lines 7–15, hundred data packages are received by Hardware and forwarded to the master. The point in time of these events is logged in

Line 10. Additionally, the name of the event ("HW?Data")[14] and the data package index are provided to the log file. These additional information are needed to achieve access to the log file entries. Thus, assuming that all data packages are forwarded correctly by the SUT, the log file must embody at least hundred log file entries at the end of this test.

Based on the timestamps recorded by the test component, real-time properties can be evaluated by real-time evaluation functions. Function evalJitter in Figure 6.20 specifies the offline evaluation function for jitter. This function gets the name of the time-critical message, the maximum index number of the data package as well as the constraint bound of the jitter.

In the beginning, the function retrieves the log file with the recorded time-critical events in Line 6. Following the mathematical formula provided on Page 79 in Section 4.4.3, jitter can be evaluated. To do so, the log file is first sorted by the name of the time-critical message by means of the **first** operator. The matching entry is retrieved in Lines 9–11. Further matching entries are selected and gathered in Lines 13–14. According to the formula, delta is calculated first (Lines 9–20). Afterwards, jitter is evaluated (Lines 23–42). There, real-time test verdict is assigned to conf_ if the jitter does not meet the predefined real-time property value (Line 34–37).

In a U2TP model, the execution of the modeled test scenarios are specified in a test control diagram. Accordingly, the test control diagram can be mapped to *TIMED*TTCN-3 module control part. References invocating test cases in a test control diagram become test case execution calls in *TIMED*TTCN-3. Furthermore, decisions in the test control diagram are expressed in decision statements, e.g. if-statement, for-statement. Figure 6.21 shows the module control part of the *TIMED*TTCN-3 test suite. In the beginning, test case TestRoaming_noWarning is executed. If it passes, the second test case TestRoaming_DataSending is run, evaluating a jitter restricted to 1.0 second in the same test run (Line 5–6).

```
1   control {
2       if (execute(TestRoaming_noWarning())==pass)
3               // Execute functional test case
4       {
5           execute(TestRoaming_DataSending(),NewYork)
6                           [ evalJitter ("HW?Data", 100, 1.0)];
7           // Execute test case with offline real−time evaluation function
8       }
9   }
```

Figure 6.21: Control Part for the Bluetooth Roaming Test Module

[14] "HW?Data" indicates that the hardware component HW has received a data package. Symbol "?" denotes a message reception.

6.5 Concluding Remarks

In this chapter, the test development process is completed by test code generation from test models. For that, TTCN-3 is chosen as an approved standardized language for system tests. To enable real-time test execution with TTCN-3, a real-time extension is introduced. In order to achieve a better tool support and higher reusability of TTCN-3, we proposed only few changes to the test language: By means of absolute time, timestamps can be collected and the absolute execution time of an event can be resumed. Timestamps can be evaluated during or after a test run. For the offline evaluation after the test run, *TIMED*TTCN-3 offers a flexible logging mechanism. Furthermore, *TIMED*TTCN-3 can also be used for distributed test architectures, since it supports synchronization of test components.

*TIMED*TTCN-3 is a testing language used to implement, execute and evaluate functional as well as real-time tests. *TIMED*TTCN-3 is downwardly compatible to TTCN-3 and hence profits from existing TTCN-3 test infrastructures. Of course, real-time test features still have to be instrumented and system specific information need to be added to the abstract *TIMED*TTCN-3 test suites to make them executable across different platforms. However, the *TIMED*TTCN-3 runtime system benefits from support from the TTCN-3 interfaces TRI and TCI whereas the TRI platform adaptor provides access to read the local clock of a test component. The TCI enables logging facilities and test component creations. Thus, log file handling and the timezone concept of *TIMED*TTCN-3 may benefit from an enhanced TCI specification. Moreover, an Extensible Markup Language (XML) format for log files improves an automated access to the log file content [W3C04].

A comparison to *TIMED*TTCN-3 concepts confirms the suitability of the selected definitions in the U2TP. Mapping rules allow the generation of *TIMED*TTCN-3 code based on U2TP models. These rules extend the existing mapping table recommended in the U2TP standard with respect to *TIMED*TTCN-3 real-time concepts. Concepts like timezones and log file which cannot be mapped to TTCN-3 can be realized in *TIMED*TTCN-3. The use of *TIMED*TTCN-3 and the applicability of the mapping rules are demonstrated in the Bluetooth roaming case study.

Chapter 7

Summary and Outlook

"Prediction is very difficult, especially about the future!" – *Niels Bohr*

In this last chapter, the work presented in this document is summarized. An outlook concerning future works on MDT can be found in Section 7.2. We complete our work with some closing words about MDA and MDT.

7.1 Summary

In this work, an approach to Model-Driven Testing based on the MDA Framework has been presented. The main idea of this test development approach is to derive test models from existing system models and finally generate executable test code from the test model. Standardized technologies and languages like MOF 2.0, UML 2.0, QVT and TTCN-3 are utilized in the approach. The following results are achieved:

To enable test modeling, UML 2.0 has been extended by various test concepts to model test architecture, test behavior, test data and time. These concepts are specified as stereotypes in a UML 2.0 profile called ***UML 2.0 Testing Profile (U2TP)***. Mapping between U2TP and executable test languages, such as JUnit and TTCN-3, facilitate the execution of test models.

Since 2004, U2TP has become an official OMG standard [U2T04]. As a consortium member, the author has participated in defining various test concepts, with emphasis on the specification of the test architecture, test behavior and time concepts. Also, contributions have been made to the definition of U2TP and TTCN-3 mapping rules.

Originally, the U2TP standard does not propose how to use the profile for test model specification. In order to close this gap, a ***guideline to U2TP*** to utilize U2TP concepts is presented and test models from existing UML 2.0 system models are derived. Doing so, the U2TP concepts are also prioritized and categorized according to their importance.

Following the guideline, ***transformation rules*** are defined to enable automatic generation of test models from system models. These rules are implemented in a model transformation language called *Tefkat*.

In order to incorporate test-specific requirements during transformation, a MOF 2.0-based metamodel called ***Test Directives metamodel*** has been defined. The Test Directives metamodel provides valuable means for test scenario selection and specification, default specification and activation, timer definition, configuration setting and other test-specific information.

The intelligence of the test model transformation is comprehended in the Test Directives model and the transformation rules themselves. As a result of this transformation, test model skeletons are generated. Their completeness and complexity depend on those of its initializing system model. To accomplish the test model specification, further refinements can be made.

Mapping U2TP concepts to executable test code enables the execution of the models. TTCN-3 is a standardized testing language providing concepts for functional and conformance testing. To enable both functional and real-time testing, a real-time extension of TTCN-3 called ***TIMED*TTCN-3** has been developed. In *TIMED*TTCN-3, concepts of absolute time, means to specify clock-synchronised test components, extensions to existing logging mechanisms, support of online and offline tests evaluations, and a new test verdict to existing TTCN-3 verdicts have been proposed. By means of adapters, *TIMED*TTCN-3 code can run on different platforms. The *TIMED*TTCN-3 concepts have been proposed[1] and partially adopted to the newest TTCN-3 standard document[2].

As a final step in this MDT development process, the existing ***mapping table*** of U2TP and TTCN-3 as defined in the standard has been augmented by real-time concepts of *TIMED*TTCN-3 to allow functional and real-time test execution[3].

To demonstrate the relevance and applicability of the presented MDT approach, the author utilized a Bluetooth roaming application as a case study where, starting from a UML 2.0 system model, a U2TP model has been derived by means of the guideline and transformation rules, and subsequently generated *TIMED*TTCN-3 code by the mapping rules.

[1] *TIMED*TTCN-3 and its graphical format *TIMED*GFT have been documented and submitted to ETSI as change requests for TTCN-3 and GFT in [Neu02, Dai03].

[2] In the current TTCN-3 standard, an extended logging mechanism can be found and the provision of logs is achieved by the TTCN-3 Logging Interface (TLI) [ETS05f].

[3] In case that other executable code shall be generated, e.g. Java or C++, new mapping rules must be defined and appropriate test engines with execution environment are needed.

7.2 Outlook

The approach presented in this work is just a step towards Model-Driven Testing and it is impossible to cover all aspects involved in MDT in a single PhD thesis. Thus, further investigations are needed to improve the test development process.

When working on this test model transformation approach, the author also considered taking a CIM[4] as the initial system model. However, since the quality of the outcoming test model skeleton depends on the granularity of the initial system model, a CIM is too vague and thus is not a good candidate for the presented transformation approach. Still, since a CIM already defines some aspects of system requirements, it might be reasonable to start test development on that stage in some other MDT approaches.

In this work, UML 2.0 is used for system modeling. However in practice, UML 2.0 is often used as a DSL by profiling UML 2.0 [Coo04]. To enable transformation and derivation from profiled system models, the transformation rules introduced in this work must be modified and aligned to its metamodel structure. Nevertheless, the transformation ideas introduced in this work can be adopted. Experiences have shown that the guideline and transformation ideas introduced in this work are applicable for profiled UML 2.0 system models. Industrial and research projects at the Fraunhofer FOKUS like [BDC+06, LN05] have given proof.

The Test Directives metamodel introduced in this work provides good means to integrate test requirements to the test model during transformation. However, it does not address test strategies such as coverage criterias, boundary value analysis or negative tests. To do so, an even deeper insight into the system specification is required. Also, systematic test data analysis can be used which can be achieved by e.g. Classification Tree Method (CTM) [Gro95]. An approach to integrating CTM with TTCN-3 for automatic test data generation can be found in [DDB+05], where an Eclipse plugin is implemented. A similar approach can be developed to integrate CTM with U2TP. Furthermore, a Graphical User Interface (GUI) to create a Test Directives model would improve user-friendliness during the test development process. Some work on implementing a GUI in Eclipse has already started in [Pia06]. In principle, the Test Directives metamodel specification could be improved in order to provide general means and more flexibility to its user, e.g. grouping of classifiers and properties.

*Timed*TTCN-3 extends TTCN-3 by only few changes and is downwardly compatible to TTCN-3. Hence, existing test suites may be reused and, e.g. instrumented for real-time test features. In this work, formal semantics of

[4]MDA introduces an even more abstract level of models than PIM, called CIM. It is a view of a system from the computation independent viewpoint. It is a domain model or business model and does not show details of the structure of systems [OMG03].

the new real-time constructs in *Timed*TTCN-3 are not addressed. Most extensions can be explained by an extension of the existing TTCN-3 semantics. In fact, only the concepts of absolute time and the **resume** operation require new semantics. Also, load testing with load generation and distribution is an interesting area for further reseach. To do so, it is necessary to establish some background load to obtain a realistic environment by e.g. an external load generator. Some related work on load testing and load distribution with TTCN-3 can be found in [Din04, SDA05].

Last but not least, further survey on transformation rules with different case studies are needed. The transformation rules introduced in this work concentrate mainly on UML 2.0 diagram types such as Class Diagrams, Composite Structure Diagrams, Object Diagrams, Interaction Diagrams and State Machines. Activity Diagrams or Use Case Diagrams are not covered in this approach at all and need further effort. Some effort with Activity Diagrams are already made in projects like [BDC+06]. A further interesting topic on MDT is to test the transformation rules themselves. Fleurey et al. bring up such a discussion in [FSB04].

7.3 Closing Words

The opinion about MDA is quite controversial. While some people embrace the idea of MDA with much enthusiasm, other people debate that MDA is just an old idea with a new name. Typically, Computer-Aided Software Engineering (CASE) has had the same idea of using models for code generation in the '80s and failed to live up to its promises. Anyhow, many people see the main failure of CASE due to the fact that tool vendors were unable to agree on a common approach and to develop supporting standards [CG04].

MDA has learned the lesson of CASE and thus defines its languages and technologies as standards to achieve commonality of its approach. Certainly, MDA has not yet reached its maturity as it promises and some definitions within MDA are still vague. But the work on MDA is still at its pre-mature stage and a lot more effort are required. Though, MDA has already reached a surprisingly high resonance and acceptance in both the industrial and research worlds.

We think that MDA is **not a revolution** like the Internet has been to the mankind. But it is **an evolution** towards better software engineering to enhance productivity, portability and interoperability of software. We hope that with the approach to MDT introduced in this PhD thesis, another small step towards software quality guarantee is made!

Glossary

Model: A model of a system is the specification of the system and its environment. A model is often presented as a combination of diagrams and text.

Model-Driven: The MDA is an approach to system development using the power of models during software development. It is model-driven because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance and modification.

Platform: A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.

Refinement: Model refinement is a minor transformation step. It describes the relationship between models of the same abstraction level regarding the platform.

Software Development Process: The software development process can be divided in *system development process* and *test development process*. In the former, a system is created and implemented throughout different development steps. In the latter, the test evaluating the system is developmed.

System Model: A test model is a visualization of a system incorporating system relevant information.

Test Model: A test model is a visualization of a system incorporating test-specific artefacts.

Transformation: Model transformation is the process of converting one model to another model of the same system.

Acronyms

ASN.1 Abstract Syntax Notation One

API Application Programm Interface

ASL Action Semantics Language

ATL Atlas Transformation Language

BNF Backus Naur Form

CASE Computer-Aided Software Engineering

CIM Computation Independent Model

CL Core Language

cMOF complete MOF

CWM Common Warehouse Metamodel

CTMF Conformance Testing Methodology and Framework

CTM Classification Tree Method

DSTC Distributed System Technology Centre

DSLs Domain-Specific Languages

EDOC Enterprise Distributed Object Computing

EMF Eclipse Modelling Framework

eMOF essential MOF

ETSI European Telecommunications Standards Institute

FIFO First In First Out

FSM Finite State Machine

GFT Graphical Presentation Format for TTCN-3

GUI Graphical User Interface

IDL Interface Definition Language

ITU-T Telecommunication Standardization Sector of ITU

IUT Implementation Under Test

LTS Labelled Transition System

MDA Model-Driven Architecture

MDT Model-Driven Testing

MOF 2.0 Meta-Object Facility, version 2

MOF Meta-Object Facility

MSC Message Sequence Chart

MSC-2000 Message Sequence Chart 2000

MTBF Mean Time Between Failure

MTF Model Transformation Framework

MTC Main Test Component

OCL Object Constraint Language

OS Operational Semantics

OMG Object Management Group

PM Platform Model

PIM Platform-Independent Model

PSM Platform-Specific Model

PerfTTCN Performance TTCN

PTC Parallel Test Component

QoS Quality of Service

QVT Queries/Views/Transformations

RT-TTCN Real-Time TTCN

RFP Request For Proposal

RUP Rational Unified Process

SDL Specification and Description Language

SUT System Under Test

TCI TTCN-3 Control Interface

TLI TTCN-3 Logging Interface

TFW Test Framework

TFT Tabular Presentation Format for TTCN-3

TOTEM Testing Object-orienTed systEms with the Unified Modelling Language

TRI TTCN-3 Runtime Interface

TTCN Tree and Tabular Combined Notation

TTCN-2 Tree and Tabular Combined Notation, version 2

TTCN-3 Testing and Test Control Notation, version 3

UML 2.0 Unified Modeling Language, version 2

UML Unified Modeling Language

U2TP UML 2.0 Testing Profile

V-Model XT V-Model eXtreme Tailoring

XML Extensible Markup Language

XMI XML Metadata Interchange

Bibliography

[ANM97] A. Alonistioti, G. Nikolaidis, and I. Modeas. SDL-based modelling and design of IN/UMTS handover functionality. In A. Cavalli and A. Sarma, editors, *SDL '97: Time for Testing - SDL, MSC and Trends*. Elsevier, 1997.

[AO00] A. Abdurazik and J. Offutt. Using UML Collaboration Diagrams for Static Checking and Test Generation. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 383–395. Springer, 2000.

[ARFC03] A. Andrews, S. Ghosh R. France, and Gerald Craig. Test Adequacy Criteria for UML Design Models. *Software Testing, Verification and Reliability*, 13(2):95–127, 2003.

[ATL06] 2006. http://www.sciences.univ-nantes.fr/lina/atl.

[ATM99a] ATM Forum Performance Testing Specification (AF-TEST-TM-0131.000). The ATM Forum Technical Committee, 1999.

[ATM99b] ATM Forum Traffic Management Specification Version 4.1 (AF-TM-0121.000). The ATM Forum Technical Committee, 1999.

[ATM00] ATM Forum UNI Signalling Performance Test Suite (AF-TEST-0158.000). The ATM Forum Technical Committee, 2000.

[Bal98] H. Balzert. *Lehrbuch der Software-Technik*, volume 2. Spektrum Akademischer Verlag, 1998.

[BBJ+02] P. Baker, P. Bristow, C. Jervis, D. King, and B. Mitchell. Automatic Generation of Conformance Tests From Message Sequence Charts. In *Proceedings of the 3rd SAM (SDL and MSC)*

Workshop – Telecommunications and beyond: The Broader Applicability of SDL and MSC, volume 2599 of *Lecture Notes in Computer Science (LNCS)*. Springer, June 2002.

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview, 1994.

[BDC+06] M. Busch, Z. R. Dai, R. Chaparadza, A. Hoffmann, L. Lacmene, T. Ngwangwen, G. C. Ndem, D. Serbanescu, I. Schieferdecker, and J. Zander-Nowicka. Model Transformer for Test Generation from Test Models, September 2006. Submitted to CONQUEST 2006 – 9th International Conference on Quality Engineering in Software Technology.

[BDG+04] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, S. Lucio, E. Samuelsson, I. Schieferdecker, and C. E. Williams. The UML 2.0 Testing Profile. In *8th Conference on Quality Engineering in Software Technology 2004 (CONQUEST04)*, Nuremberg, Germany, September 2004.

[Bei95] B. Beizer. *Black-Box Testing*. John Wiley & Sons, Inc, 1995.

[Bez01] J. Bezivin. From Object Composition to Model Transformation with the MDA. In *IEEE TOOLS-39*, Santa Barbara, USA, August 2001.

[Bin99] R. Binder. *Testing Object-Oriented Systems, Models, Patterns, and Tools*. Addison Wesley, 1999.

[BJK+05] M. Broy, B. Jonsson, J.-P. Karoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*. LNCS 3472. Springer, 2005.

[BL01] L. Briand and Y. Labiche. A UML-Based Approach to System Testing. In *Proceedings of the 4th International Conference on the UML (Lecture Notes in Computer Science)*, Toronto, Canada, 2001. Springer.

[Blu04] Bluetooth Special Interest Group. *Specification of the Bluetooth System (version 2.0)*, November 2004. http://www.bluetooth.com.

[Blu06] 2006. http://www.iti.uni-luebeck.de/Research/MUC/EKG/.

[BSKH04] M. Born, I. Schieferdecker, O. Kath, and C. Hirai. Combining System Development and System Test in a Model-Centric Approach. In *RISE 2004 – International Workshop on Rapid Integration of Software Engineering Techniques*, Luxembourg, November 2004.

[BSL00] M. Born, I. Schieferdecker, and M. Li. UML Framework for Automated Generation of Component-Based Test Systems, 2000.

[BSM+03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework: A Developer's Guide*. Addison Wesley, August 2003.

[Car03] K. Carter. Initial Submission to MOF Query/Views/Transformations RFP. OMG Document: ad/03-03-11, March 2003.

[Cav01] A. Cavarra. AGEDIS Language Specification. Project Deliverable 2.2. The AGEDIS project, 2001. http://www.agedis.de.

[CG04] S. Cook and M. Guttman. *The MDA Journal*, chapter 5 - 8: The Cook-Guttman Debate on MDA and Microsoft. Meghan-Kiffer Press, 2004.

[CH03] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, USA, October 2003.

[CKLY98] J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. Deriving Petri Nets from Finite Transition Systems. In *IEEE Transactions on Computers*, volume 47, pages 859–882, 1998. citeseer.ist.psu.edu/article/cortadella98deriving.html.

[Con06] Eclipse Consortium. Eclipse UML2 Plugin, 2006. http://www.eclipse.org/uml2/.

[Coo04] S. Cook. *The MDA Journal*, chapter 5: Domain-Specific Modeling and Model Driven Architecture. Meghan-Kiffer Press, 2004.

[CS03] M. Jesse Chonoles and J. A. Schardt. *UML 2 for Dummies*. Wiley & Sons, 2003.

[Dai03] Z. R. Dai. TTCN-3 Change Request No. 232. Submitted to European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), June 2003.

[Dai04a] Z. R. Dai. Model-Driven Testing with UML 2.0. In *Proceedings of Second European Workshop on Model Driven Architecture (MDA) EWMDA*, Canterbury, England, September 2004.

[Dai04b] Z. R. Dai. UML 2.0 Testing Profile. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, chapter 16. Dagstuhl, Dagstuhl, 2004.

[DDB+05] Z. R. Dai, P. Deussen, M. Busch, L. L. Pianta, T. Ngwengwen, J. Herrmann, and M. Schmidt. Automatic test data generation for TTCN-3 with CTE. In *Proceedings of ICSSEA*, Paris, November 2005.

[Dep04] Deployment and Configuration of Component-based Distributed Applications Specification. OMG, 2004.

[DGL+00] K. Duddy, A. Gerber, M. Lawley, K. Raymond, and J. Steel. Declarative Transformation for Object-Oriented Models, 2000.

[DGL+03a] K. Duddy, A. Gerber, M. Lawley, K. Raymond, and Jim Steel. Model Transformation: A declarative, reusable patterns approach. In *7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, pages pp. 174–185, 2003.

[DGL+03b] K. Duddy, A. Gerber, M.J. Lawley, K. Raymond, and J. Steel. Modelware for Middleware. In *1st Workshop on Model-driven Approaches to Middleware Applications Development*, MAMAD 2003, Rio de Janeiro, Brazil, June 2003.

[DGN02] Z. R. Dai, J. Grabowski, and H. Neukirchen. Timed TTCN-3 – A Real-Time Extension for TTCN-3. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems*, volume 14, Berlin, March 2002. Kluwer.

[DGN03] Z. R. Dai, J. Grabowski, and H. Neukirchen. TimedTTCN-3 Based Graphical Real-Time Test Specification. In D. Hogrefe and A. Wiles, editors, *Testing of Communicating Systems*, volume 2644 of *Lecture Notes in Computer Science (LNCS)*. Springer, May 2003.

[DGNP03] Z. R. Dai, J. Grabowski, H. Neukirchen, and H. Pals. UML-Based Testing of Roaming with Bluetooth Devices. In *Proceedings of the First Hangzhou-Lübeck Workshop on Software Engineering*. University of Hangzhou, China, November 2003.

[DGNP04] Z. R. Dai, J. Grabowski, H. Neukirchen, and H. Pals. From Design to Test with UML – Applied to a Roaming Algorithm for Bluetooth Devices. In R. Hierons, editor, *Testing of Communicating Systems*, volume 2978 of *Lecture Notes in Computer Science (LNCS)*. Springer, May 2004.

[Din04] G. Din. TTCN-3. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, chapter 15. Dagstuhl, Dagstuhl, 2004.

[DST04] S. Dibuz, T. Szabó, and Zsolt Torpis. BCMP Performance Test with TTCN-3 Mobile Node Emulator. In R. Hierons, editor, *Testing of Communicating Systems*, volume 2978 of *Lecture Notes in Computer Science (LNCS)*. Springer, May 2004.

[DW99] W. Dröschel and M. Wiemers. *Das V-Modell 97, Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz*. Oldenburg Verlag, 1999.

[Ecl06] 2006. http://www.eclipse.org/.

[EGH+97] A. Ek, J. Grabowski, D. Hogrefe, R. Jerome, B. Koch, and M. Schmitt. Towards the Industrial Use of Validation Techniques and Automatic Test Generation Methods for SDL Specifications. In A. Cavalli and Sarma A, editors, *SDL '97: Time for Testing - SDL, MSC and Trends*. Elsevier, 1997.

[EHS97] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL, Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.

[ETS98] ETSI European Guide (EG) 202 103: Guide for the use of the second edition of TTCN (Revised version). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), 1998.

[ETS02] ETSI European Standard (ES) 201 873-1 V2.2.1 (2002-08). The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), also published as ITU-T Recommendation Z.140, 2002.

[ETS03] ETSI European Standard (ES) 201 873-5 V1.1.1 (2003-02). The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), 2003.

[ETS05a] ETSI European Standard (ES) 201 873-1 V3.1.1 (2005-06). Methods for Testng and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), Reference: RES/MTS-00090-1, June 2005.

[ETS05b] ETSI European Standard (ES) 201 873-2 V3.1.1 (2005-06). Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 2: the Tree and

Tabular Combined Notation (TFT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), Reference: RES/MTS-00090-2 ttcn3 tft, June 2005.

[ETS05c] ETSI European Standard (ES) 201 873-3 V3.1.1 (2005-06). Methods for Testing and Specification (MTS); The Tree and Tabular Combined Notation version 3; Part 3: Graphical Presentation Format for TTCN-3 (GFT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), Reference: RES/MTS-00090-3 ttcn3 gft, 2005.

[ETS05d] ETSI European Standard (ES) 201 873-4 V3.1.1 (2005-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), Reference: RES/MTS-00090-4 ttcn3 os, 2005.

[ETS05e] ETSI European Standard (ES) 201 873-5 V3.1.1 (2005-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), Reference: RES/MTS-00090-5 ttcn3 tri, 2005.

[ETS05f] ETSI European Standard (ES) 201 873-6 V3.1.1 (2005-06). Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), Reference: RES/MTS-00090-6 ttcn3 tci, 2005.

[FP04] D.S Frankel and J. Parodi. *The MDA Journal*. Meghan-Kiffer Press, 2004.

[Fra03] D. S. Frankel. *Model Driven Architecture - Applying MDA to Enterprise Computing*. OMG Press, 2003.

[Fre06] 2006. http://www.freertos.org/.

[FSB04] F. Fleurey, J. Steel, and B. Baudry. Validation in Model-Driven Engineering: Testing Model Transformations. *Model, Design and Validation. IEEE Proceedings*, pages 29 – 40, Nov. 2004.

[FSG06] R. B. France and T. Dinh-Trong S. Ghosh. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *IEEE Computer*, Model Driven Software Development:59–67, February 2006.

[GH02] J. Grabowski and D. Hogrefe. SDL and MSC-Based Specification and Automated Test Case Generation for INAP. *Telecommunication Systems*, 20(3,4):265–291, July 2002.

[GKS00] R. Gecse, P. Kremer, and J. Zoltan Szabo. HTTP Performance Evaluation with TTCN. In Hasan Ural, Rober Probert, and Grepor Bochmann, editors, *Testing or Communication Systems: Tools and Techniques*. Ericsson Budapest, Kluwer Academic, 2000.

[GKSH99a] J. Grabowski, B. Koch, M. Schmitt, and D. Hogrefe. SDL and MSC Based Test Generation for Distributed Test Architectures. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 - The next Millenium*. Elsevier Science Publishers B.V., June 1999.

[GKSH99b] J. Grabowski, B. Koch, M. Schmitt, and D. Hogrefe. SDL and MSC Based Test Generation for Distributed Test Architectures (http://www.itm.mu-luebeck.de/public/grabowski_d.html). In R. Dssouli, G. v. Bochmann, and Y. Lahav, editors, *SDL'99 - The next Millennium*. Elsevier, June 1999.

[GR03] A. Gerber and K. Raymond. MOF to EMF: There And Back Again. In *OOPSLA 2003*, Anaheim. USA, Oct 2003. Proc. Eclipse Technology Exchange Workshop.

[Gra02] J. Grabowski. *Specification Based Testing of Real-Time Distributed Systems*. Habilitation thesis, Universität zu Lübeck, 2002.

[Gro95] J Grochtmann, M.; Wegener. Test Case Design Using Classification Trees and the Classification-Tree Editor CTE. In *8th International Software Quality Week*, San Francisco, USA, May 1995.

[GSDH97] J. Grabowski, R. Scheurer, Z. Dai, and D. Hogrefe. Applying SAMSTAG to the B-ISDN protocol SSCOP. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communication Systems*, volume 10. Chapman & Hall, 1997.

[GV03] C. Girault and R. Valk. *Petri Nets for System Engineering – A Guide to Modeling, Verification, and Applications* . SPRINGER, BERLIN, Jan 2003.

[GWWH00] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe. On the Design of the New Testing Language TTCN-3. In H. Ural,

R.L. Probert, and G. von Bochmann, editors, *Testing of Communicating Systems*, volume 13. Kluwer Academic Publishers, 2000.

[HKN01] D. Hogrefe, B. Koch, and H. Neukirchen. Some Implications of MSC, SDL and TTCN Time Extensions for Computer-aided Test Generation. In R. Reed and J. Reed, editors, *SDL2001 – Meeting UML*, volume 2078 of *Lecture Notes in Computer Science (LNCS)*. Springer, June 2001.

[HMP91] T. Henzinger, Z. Manna, and A. Pnueli. Timed Transition Systems. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory and Practice*, volume 600 of *Lecture Notes in Computer Science (LNCS)*, pages 226–251. Springer, June 1991.

[HN03] A. Hartman and K. Nagin. Model Driven Testing - AGEDIS Architecture Interfaces and Tools. In *1st European Conference on Model Driven Software Engineering*, Nuremburg, 2003.

[HN04] A. Hartman and K. Nagin. The AGEDIS Tools for Model Based Testing. In *ISTA*, 2004.

[IBM06a] IBM. Model Transformation Framework 1.0.0, 2006. http://www.alphaworks.ibm.com/tech/mtf.

[IBM06b] 2006. http://www-306.ibm.com/software/rational/.

[IBM06c] 2006. http://www-106.ibm.com/developerworks/wireless.

[IEE96] IEEE Standard 1003.1: Information technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language]. Institute of Electrical and Electronics Engineers (IEEE), 1996.

[IET90] Request for Comments 1193: Client requirements for real-time communication services. Internet Engineering Task Force (IETF), 1990.

[IET91] Request for Comments 1242: Benchmarking Terminology for Network Interconnection Devices. Internet Engineering Task Force (IETF), July 1991.

[IET98] Request for Comments 2330: Framework for IP Performance Metrics. Internet Engineering Task Force (IETF), May 1998.

[IET99] Request for Comments 2679: A One-way Delay Metric for IPPM. Internet Engineering Task Force (IETF), September 1999.

[IET02] Request for Comments 3393: IP Packet Delay Varation Metric for IP Performance Metrics (IPPM). Internet Engineering Task Force (IETF), November 2002.

[ISO97] Information Technology – Open Systems Interconnection – Conformance testing methodology and framework. ISO/IEC, 1994-1997. International ISO/IEC multipart standard No. 9646.

[IT96] ITU-T. Specification and Description Language (SDL). ITU–T Rec. Z.100, Geneva, 1996.

[ITU99] ITU-T Recommendation Z.100: Specification and Description Language (SDL). International Telecommunication Union (ITU-T), Geneve, 1999.

[JBR99] Jacobson, Booch, and Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[JUn06a] 2006. http://www.junit.org/.

[JUn06b] 2006. http://www.clarkware.com/software/JUnitPerf.html.

[KCL03] University of York King's College London. An Evaluation of Compuware OptimalJ Professional Edition as an MDA Tool, September 2003.

[Koc01] B. Koch. *Test-purpose-based Test Generation for Distributed Test Architectures*. PhD thesis, University of Luebeck, Luebeck, 2001.

[Kru00] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison Wesley, 2000.

[KSB03] O. Kath, M. Soden, and M. Born. An Open Modelling Infrastructure integrating EDOC and CCM. In *EDOC 2003*, Brisbane, Australia, September 2003.

[KWB03] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley Pub Co, April 2003.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21, 1978.

[lL90] G. le Lann. Critical issues for the development of distributed real-time computing systems. Technical Report 1274, Institut National de Recherche den Informatique et en Automatique (INRIA), Le Chesnay (France), August 1990.

[LN05] L. Lacmene and T. Ngwangwen. A UML 2.0 Design Model to Test Model Transformation Approach. Internship Report of Technical University, Berlin, Oct 2005.

[LY96] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. In *Proceedings of the IEEE*, 1996.

[Mag00] Real Time Magazine. What makes a good RTOS. In *RTOS Evaluation Program*. Real Time Magazine, 2.0 edition, 2000. http://www.realtime-info.be.

[Med06] Medini, 2006. http://www.ikv.de/pdf/medini/mediniPP.pdf.

[Mil03] Granville Miller. What's New in UML 2.0? A Borland White Paper, Dec 2003.

[MP92] M. Mouly and M. B. Pautet. *The GSM System for Mobile Communications*. Cell & Sys, Palaiseau, France, 1992.

[Mye79] G. Myers. *The Art of Software Testing*. Wiley, 1979.

[Neu02] H. Neukirchen. TTCN-3 Change Request No. 148. Submitted to European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), June 2002.

[Neu04] Helmut Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, University of Goettingen, 2004.

[OA99] J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 416–429. Springer, 1999.

[(OM05] Object Management Group (OMG). UML Profile for EDOC final adopted specification. OMG ptc/02-02-05, 2005.

[OMG02a] OMG. Request for Proposal: MOF 2.0 Query/Views/ Transformations RFP, ad/02-04-10. OMG Document, Apr 2002.

[OMG02b] OMG. *UML Testing Profile - Request For Proposal*, April 2002. OMG Document (ad/01-07-08).

[OMG03] OMG. *MDA Guide V1.0.1*, June 2003.

[OMG04] OMG. Meta Object Facility (MOF) v2.0 Core Specification, ptc/04-10-15. OMG Document, Oct 2004.

[OMG05] OMG. Meta Object Facility (MOF) 2.0 Query/Views/-Transformations Specification. Final Adopted Specification: ptc/2005-11-01, November 2005.

[OMG06] OMG Consortium, 2006. http://www.omg.org/.

[Omo06] Omondo EclipseUML Studio Product Documentation, 2006. http://www.omondo.com.

[ORM01] Architecutre Board ORMSC. Model Driven Architecture (MDA). OMG Document ormsc/2001-07-01, July 2001.

[Par91] H. Partsch. *Requirements Engineering*. Oldenburg, 1991.

[PDGN03] H. Pals, Z. R. Dai, J. Grabowski, and H. Neukirchen. UML-Based Modeling of Roaming with Bluetooth Devices. In *Proceedings of the First Hangzhou-Lübeck Workshop on Software Engineering*. University of Hangzhou, China, November 2003.

[Per03] P. Perrotta. The Virtual Clock Test Pattern, 2003. http://www.nusco.org/docs/virtual_clock.pdf.

[Pia06] L. L. Pianta. User Driven Test Generation from System Models. To be published as Diploma Thesis at Technical University Berlin, 2006.

[RB96] Jr. R. Buchanan. *The Art of Testing Network Systems*. Wiley Computer Publishing, 1996.

[RGG99] E. Rudolph, J. Grabowski, and P. Graubmann. Towards a Harmonization of UML-Sequence Diagrams and MSC. In R. Dssouli, G. v. Bochmann, and Y. Lahav, editors, *SDL'99 - The next Millenium*. Elsevier, June 1999.

[RJB05] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Addison-Wesley, 2005.

[RR01] R. Reed and J. Reed, editors. *SDL 2001: Meeting UML, 10th International SDL Forum Copenhagen*. Springer, LNCS 2078, 2001.

[RSB90] P. Ramanathan, K.G. Shin, and R.W. Butler. Fault-Tolerant Clock Synchronization in Distributed Systems. *IEEE Computer*, 23, 1990.

[RSG00] E. Rudolph, I. Schieferdecker, and J. Grabowski. Development of an MSC/UML Test Format. In J. Grabowski and S. Heymer, editors, *FBT'2000 - Formale Beschreibungstechniken für verteilte Systeme*. Shaker Verlag, Aachen, June 2000.

[RTL06] 2006. http://www.rtlinuxfree.com/.

[Sch93] W. Schütz. *The Testability of Distributed Real-Time Systems*. Kluwer, 1993.

[SDA05] I. Schieferdecker, G. Din, and D. Apostolidis. Distributed Functional and Load tests for Web services. In *International Journal on Software Tools for Technology Transfer (STTT)*. Springer, 2005.

[SDGR03] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch. The UML 2.0 Testing Profile and its Relation to TTCN-3. Testing of Communicating Systems (Editors: D. Hogrefe, A. Wiles) – 15th IFIP International Conference, TestCom2003, Sophia Antipolis, France, May 2003, Proceedings. Lecture Notes in Computer Science (LNCS) 2644, Springer, pp. 79-94, May 2003.

[SEG+98] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink – Putting SDL-based test generation into practice. In A. Petrenko and N. Yevtuschenko, editors, *Testing of Communicating Systems*, volume 11. Kluwer, 1998.

[Sel05] Bran Selic. What's New in UML 2.0. IBM Press, April 2005.

[Sif89] J. Sifakis, editor. *Automatic Verification Methods for Finite State Systems*, volume Lecture Notes in Computer Science. SPRINGER Berlin, Grenoble, France, vol.407 edition, Jun 1989.

[SL03] A. Spillner and T. Linz. *Basiswissen Softwaretest*. dpunkt.verlag, Heidelberg, 2003.

[SR90] J. Stankovic and K. Ramamritham. What is Predictability for Real-Time Systems? *Real-Time Systems*, 2(4):247–254, 1990.

[SSR97] I. Schieferdecker, B. Stepien, and A. Rennoch. PerfTTCN, a TTCN Language Extension for Performace Testing. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, volume 10. Chapman & Hall, 1997.

[SvMF+99] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman, and A. E. Howe. Generating test cases from an OO model with an

AI planning system. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'99)*, Boca Raton, FL, 1999. IEEE Computer Society Press.

[Tar06] 2006. http://www.dstc.edu.au/Research/Projects/Pegamento.

[Tef06] 2006. http://www.dstc.edu.au:8080/qvt/index.html/.

[Tes05] TTWorkbench Professional, version 1.1, 2005. http://www.testingtech.de/products/.

[TPvB97] Q. M. Tan, A. Petrenko, and G. v. Bochmann. Checking Experiments with Labeled Transition Systems for Trace Equivalence. In *IWTCS*, 1997.

[U2T04] U2TP Consortium. *UML 2.0 Testing Profile*, April 2004. http://www.omg.org/cgi-bin/doc?ptc/2004-04-02.

[UHPB03] A. Ulrich, H. Hallal, A. Petrenko, and S. Boroday. Verifying Trustworthiness Requirements in Distributed Systems with Formal Log-file Analysis. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*. IEEE, 2003.

[UML04] *Unified Modelling Language (UML) Specification : Infrastructure, Version 2.0*, November 2004. OMG Adopted Specification (ptc/04-10-14).

[UML05] *Unified Modelling Language: Superstructure, Version 2.0*, August 2005. OMG Adopted Specification (formal/05-07-04).

[V-M04] V-Modell XT, Version 1.2.0, 2004. http://ftp.uni-kl.de/pub/v-modell-xt/Release-1.2/Dokumentation/html/.

[VVP] D. Varr´o, G. Varra´o, and A. Pataricza. Designing the Automatic Transformation of Visual Languages. Science of Computer Programming.

[W3C04] Extensible Markup Language (XML) 1.0 (Third Edition). World Wide Web Consortium (W3C) Recommendation, February 2004.

[Wal01] E. Wallmüller. *Software- Qualitätsmanagement in der Praxis*. Hanser Verlag, 2001.

[WDT+05] C. Willcock, T. Deiss, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3*. Wiley & Sons, 2005.

[Wey86] E. Weyuker. Axiomatizing Software Test Data Adequacy. *IEEE Transactions on Software Engineering*, 12(12), December 1986.

[Wey88] E. Weyuker. The Evaluation of Program-based Software Test Data Adequacy Criteria. *Communications of the ACM*, 31(6), June 1988.

[WG97] T. Walter and J. Grabowski. A Proposal for a Real-Time Extension of TTCN, 1997.

[WG99] T. Walter and J. Grabowski. A Framework for the Specification of Test Cases for Real Time Distributed Systems. In *Information and Software Technology*, volume 41. Elsevier, July 1999.

[WLA03] Information technology, Telecommunications and information, exchange between systems, Local and metropolitan area networks, Specific requirements. IEEE-SA Standards Board, 2003.

[WSG98] T. Walter, I. Schieferdecker, and J. Grabowski. Test Architectures for Distributed Systems - State of the Art and Beyond. In A. Petrenko and N. Yevtuschenko, editors, *Testing of Communicating Systems*, volume 11. Kluwer Academic Publishers, 1998.

[Z1299] Recommendation Z.120: Message Sequence Charts (MSC). International Telecommunication Union (ITU-T), Geneve, 1999.