Journal of Zhejiang University SCIENCE
ISSN 1009-3095
http://www.zju.edu.cn/jzus
E-mail: jzus@zju.edu.cn

Model-based testing with UML applied to a roaming algorithm for Bluetooth devices

DAI Zhen Ru[1], GRABOWSKI Jens[2], NEUKIRCHEN Helmut[2], PALS Holger[3]
([1]*Fraunhofer Fokus, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany*)
([2]*Institute for Informatics, University of Goettingen, Lotzestrasse 16-18, 37083 Goettingen, Germany*)
([3]*Institute of Computer Engineering, University of Luebeck, Ratzeburger Allee 160, 23538 Luebeck, Germany*)
E-mail: dai@fokus.fraunhofer.de; {grabowski,neukirchen}@cs.uni-goettingen.de; pals@iti.uni-luebeck.de

Received Dec. 2, 2003; revision accepted July 2, 2004

Abstract: In late 2001, the Object Management Group issued a Request for Proposal to develop a testing profile for UML 2.0. In June 2003, the work on the UML 2.0 Testing Profile was finally adopted by the OMG. Since March 2004, it has become an official standard of the OMG. The UML 2.0 Testing Profile provides support for UML based model-driven testing. This paper introduces a methodology on how to use the testing profile in order to modify and extend an existing UML design model for test issues. The application of the methodology will be explained by applying it to an existing UML Model for a Bluetooth device.

Key words: UML 2.0, UML 2.0 Testing Profile, Re-Usability, Bluetooth, Roaming
doi:10.1631/jzus.2004.1327 **Document code:** A **CLC number:** TP391

INTRODUCTION

The Unified Modeling Language (UML) (Eriksson, 2003) is a visual language supporting the design and development of complex object-oriented systems. While UML models focus primarily on the definition of system structure and behavior, they provide only limited means for describing test objectives and test procedures. Furthermore, the growing system complexity increases the need for solid testing. Thus, in 2001, the Object Management Group issued a Request for Proposal (U2TP web-site: http://www.fokus.fraunhofer.de/u2tp/) to develop a testing profile for UML 2.0 (UML 2.0 Infrastructure Specification, OMG Adopted Specification, ptc/03-09-15; UML 2.0 Superstructure Specification, 2004, ptc/04-05-02).

A UML profile provides a generic extension mechanism for building UML models in particular domains. The UML 2.0 Testing Profile is such an extension developed for the testing domain. It bridges the gap between designers and testers by providing means for using UML both for system modelling and test specification. This allows a reuse of UML design documents for testing and enables test development in an early system development phase. Meanwhile, the UML 2.0 Testing Profile project has come to its finalization and has become an official standard of the OMG.

In this paper, we provide a methodology for applying UML 2.0 Testing Profile concepts in an existing UML design model effectively. As a case study, the methodology will be evaluated by applying it on a UML model for roaming with Bluetooth devices (Pals *et al*., 2003).

THE UML 2.0 TESTING PROFILE

The UML 2.0 Testing Profile (U2TP) provides concepts that target the development of test specifications and test models for black-box testing (Beizer, 1995). In particular, the profile introduces four concept groups covering the aspects: test architecture, test behavior, test data and time. Together, these concepts define a modelling language for visualizing, specifying, analysing, constructing and documenting the artefacts of a test system (UML 2.0 Testing Profile, Final Adopted Specification at OMG, 2004, http://www.omg.org/cgi-bin/doc? ptc/2004-04-02).

Test architecture concepts

The test architecture concept group covers concepts for specifying test components, the interfaces of and connections between test components and to the system under test.

One or more objects within a test specification can be identified as the System Under Test (SUT). Test components are defined as objects within the test system that can communicate with the SUT or other components to realize the test behavior. Test configuration is a collection of parts, representing test components, the SUT and the connections between the test components and to the SUT. A test context groups test cases with the same initial test configuration. An arbiter is a denoted test component which is responsible for the final test result calculation which derives from temporary test results. A utility part represents a miscellaneous component which helps test components to realize their test behavior. Typically, utility parts are data bases with data pools. A scheduler controls the creation and termination of test components and the interaction between test components and the arbiter.

Test behavior concepts

The test behavior concept group covers concepts for specifying actions necessary to evaluate the objective of a test. Test behaviors can be defined by any behavioral diagram of UML 2.0, i.e. as interaction diagrams or state machines. Test objectives allow the designer to express the intention of the test. A test case is an operation of a test context specifying how a set of cooperating components interact with the SUT to realize a test objective. A test case always returns a test verdict. The handling of unexpected events (e.g. wrong responses from the SUT) is eased by the specification of defaults. A default is a separate behavior which is triggered if an event is observed that is not explicitly handled by the main test case behavior. Test verdicts specify possible test results, e.g. pass, fail, or inconclusive. The definition of the verdicts originates from the OSI Conformance Testing Methodology and Framework (ISO/IEC, 1994). Pass indicates that the SUT behaves correctly for the specific test case. Fail describes that the test case has been violated. Inconclusive is used where neither a Pass nor a Fail can be given. A validation action can be performed by a test component to denote that the arbiter is informed of a test result which was determined by that test component. During the execution of a test case a test trace is generated. A test log is used to log entries during the execution for further analysis.

Test data concepts

The test data concept group covers concepts for describing communication data between the SUT and the test components. Wildcards are useful for test data specifications, especially for data reception. U2TP introduces wildcards allowing the specification of: Any value (1…*n*) and Any or None values (0…*n*). Data Pool is a container for possible test data for the specified test. Logical partitions are used to define value sets within test parameters. Data Selection is an operation to retrieve appropriate test data. The specification of coding rules allows the user to define which encoding of data values is used in the implementation.

Time concepts

The time concept group covers concepts to constrain and control test behavior with regard to time. Timers are utilized to manipulate and control test behavior, as well as to ensure the termination of test cases. Time zones are defined to group components within a distributed system and allow the comparison of time events within the same time zone.

A METHODOLOGY FOR U2TP

The U2TP has just been developed at the Object Management Group (OMG). For a tester who uses the U2TP for the first time, it is hard to see which concepts are important for his/her test specifications and which concepts are less important. In this section, we will explain briefly how a tester can apply the concepts of the U2TP effectively after having received a detailed design model which should be tested. To clarify the terminologies: With design model, we mean the system design model in UML. When talking about the test model, we mean the UML model enriched with U2TP concepts.

Having a design system model, the tester may want to specify tests for the system. For that, the existing design model can be enriched with U2TP concepts. The following aspects must be considered when transforming a design model into a test model:

First of all, define a new UML package as the test package of the system. Import the classes and interfaces from the system design package in order to get access to message and data types in the test specification.

Next, start with the specification of the test architecture and continue with test behavior specifications. Test data and time are usually comprised in either the test architecture (e.g. timezone or data pool) or test behavior (e.g. timer or data partitioning) specifications.

Below, issues regarding test architecture and test behavior specifications are listed. They are subdivided into two categories: mandatory issues and optional issues. Mandatory issues can normally be retrieved directly from the design model, while optional issues are specific to test requirements and therefore, can seldom be retrieved from existing UML diagrams. However, they are also not always needed for the test model. The most important issues are the specification of the SUT components, the test components, the test cases and the verdict settings:

Test architecture

1. Mandatory

(1) Which system component/components would you like to test? Assign it/them to System Under Test (SUT).

(2) Depending on their functionalities, test components have to be defined. Try to group the system components (except the SUT) to test components.

(3) Specify a test context class listing the test attributes and test cases, also possible test control and test configuration.

2. Optional

(1) In order to define the ordering of test case execution, specify the test control. The simplest form is to execute test cases sequentially. In more complex test controls, loops and conditional test execution can be specified.

(2) Test configuration can be easily retrieved by means of existing interaction diagrams: Whenever two components exchange messages with each other, assign a communication channel between the components. If there is no interaction diagram defined in the design model, connect the test components and SUT to an appropriate test configuration so that the configuration is relevant for all test cases included in the test context.

(3) Determine utility parts within the test configuration.

(4) Determine an arbiter for verdict arbitration.

(5) Assign timezones to the components. Timezones are normally needed if a distributed test system is built and time values of different components need to be compared.

(6) Look at coding rule specifications.

Test behavior

1. Mandatory

(1) For designing the test cases, take the given interaction diagrams of the design model and change (i.e. rename or group) the instances and assign them

with stereotypes of the U2TP (i.e. test component or SUT) according to their functionalities.

(2) Assign verdicts at the end of each test case specification. Usually, the verdict in a test case is set to pass.

2. Optional

(1) Specify default behaviors using wildcards for setting a fail or inconclusive verdict.

(2) Define time events by means of timers or time constraints.

A CASE STUDY: ROAMING WITH BLUETOOTH DEVICES

In this section, we will provide an example on how to design tests and modify an existing design model to obtain a test model. As a case study, we take the UML design model for roaming with Bluetooth devices which is introduced in (Pals *et al.*, 2003). For the model modification, we will apply step by step the methodology introduced in the previous section. The main focus of this case study is to show that classes and interfaces specified in the design model can be re-used in the test model (Dai *et al.*, 2004).

Test preparation

Before amending the design model, the focus of the test must be defined, i.e. which classes should be tested and which interfaces does the tester need in order to get access to these classes. For our case study, the functionalities of the SlaveBTRoaming layer[1] are subject of test.

Fig.1[2] presents the test configuration with one slave and two masters. The classes originate from the BluetoothRoaming package of the design model (Pals *et al.*, 2003): The focus of our tests is the SlaveBTRoaming layer. Thus, the SlaveApplication layer is one test component. Other test components are the underlying Bluetooth Hardware layer and the master components Master1 and Master2.

On the top of the slave and the masters, a new test component of class Test-Coordinator is specified. This test component is the main test component which administrates and instructs the other test components during the test execution. The coordinator is also responsible for the evaluation of the test cases and the setting of verdicts during test case execution. The coordinator has access to the utility part Location-DataBase. This data base embodies the LocationServer, which owns the slave roaming lists and the network structure table used for roaming decisions. Communication between the Test-Coordinator and the masters is performed via the Test Coordination Interface (TCI).

This test configuration is very flexible: The Bluetooth Hardware layer used in a test configuration might either be real Bluetooth (i.e. consisting of the slave's Bluetooth hardware SlaveBT-HW and the master's Bluetooth hardware MasterBT-HW) or emulated by software. Moreover, different multi party test configurations can easily be obtained by adding more masters. The master test component can be regarded as sub-divided into a Master Roaming and Master Application layer. This allows re-use of all the classes specified in the design model. Additionally, in a different test stage,

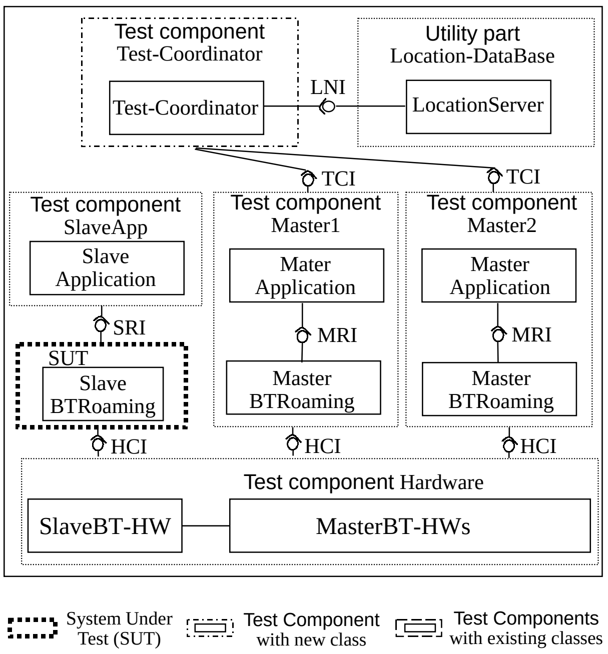


Fig.1 Role Assignment for System Components

[1]Layer is a term used in the context of communication protocols. In this paper, we will use it as a synonym to component within an object-oriented system.

[2]This diagram is not a UML diagram.

it would be possible to replace more and more of the emulated test components with real implementations. Consequently, it is easy to perform integration tests with such a test configuration as well.

In our case study, the following functionalities of the SlaveRoaming layer should be tested:

(1) Is the SlaveRoaming layer able to choose a new master by looking up in its roaming list when the connection with its current master gets weak?

(2) Does the SlaveRoaming layer request a connection establishment to the chosen master?

(3) Does the SlaveRoaming layer wait for a connection confirmation of the master when the connection has been established?

(4) Does the SlaveRoaming layer send a warning to the environment, when no master can be found and the roaming list is empty?

These test objectives assume that basic functionalities of the SlaveRoaming layer like data forwarding from the application layer to the hardware layer have already been tested in a preceding capability test.

Test architecture specification

First of all, a test package for the test model must be defined. Our package is named BluetoothTest (Fig.2a). The test package imports the classes and interfaces from the BluetoothRoaming package (Pals *et al*., 2003) in order to get access to the classes to be tested.

In the test preparation phase in the section on test preparation, we have assigned the SlaveBTRoaming layer to SUT and other system components to test components. The test package consists of five test component classes, one utility part and one test context class. The test context class is called BluetoothSuite. It contains various test attributes, some test functions and test cases (Fig.2b).

Test configuration and test control are also specified in the test context class. The test configuration (Fig.3a) corresponds with the test configuration in Fig.1, except that it consists of one slave and four masters m1–m4. Ports with interfaces connect the test components and the SUT to each other.

Fig.3b illustrates the test control, indicating the execution order of the test cases: First, test case TestRoaming_noWarning is executed. If the test result is pass, the second test case TestRoaming_withWarning will also be executed. Otherwise, the test is finished.

Test behavior specification

The test cases which will be shown were all derived from the sequence diagrams, state machines and activity diagrams of the design model. Only little effort was necessary for deriving the test case specification. Some of the test cases may also be generated automatically.

In the section on test preparation, we have listed the test objectives of the case study. As an example, we will present a test case with the following scenario:

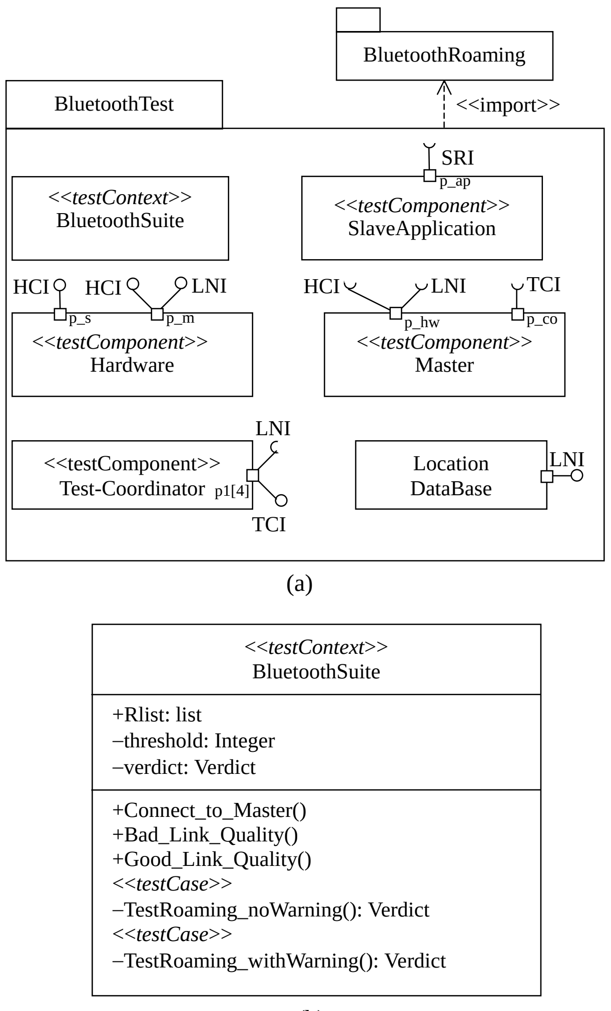
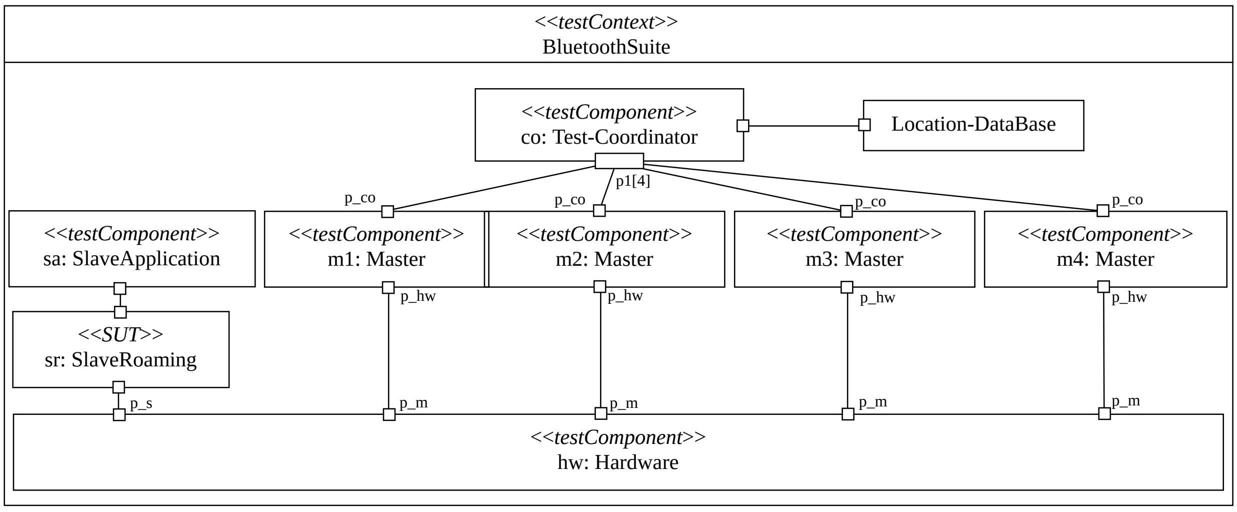
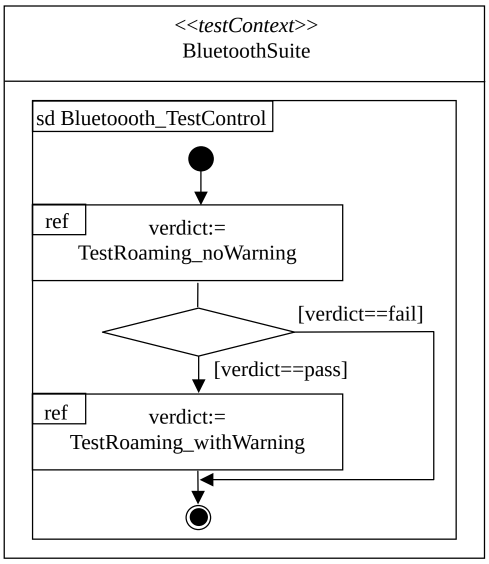


Fig.2 Test packages (a) and test context class (b)



(a)



(b)

Fig.3 Test configuration (a) and test control (b)

After the exchange of two data packages, the link quality between Slave and its current 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.

Fig.4 depicts the test case for the scenario given above. Test case TestRoaming_NoWarning starts with the activation of the timer T1 with a duration of six seconds. T1 is a guarding timer which is started at the beginning and stopped at the end of a test case. It assures that the test finishes properly even if e.g. the SUT crashes and does not respond anymore. In this case, the timeout event is caught by a default behavior.

The function Connect_To_Master referenced at the beginning of the test case establishes a connection between the Slave and Master m1 Fig.5a): The connection request (con_request) is initiated by the SlaveApplication and is forwarded to the master. The master informs the Test-Coordinator about that observation. Then, the master accepts the connection (con_accept), resulting in a confirmation sent from the Bluetooth hardware to both the slave and the master. Thereupon, the master informs the Test-Coordinator about the successful connection, which allows the Test-Coordinator to build a new roaming list containing the masters (reference makeList) and to transfer it via the master to the slave using the message roamingList ([M2, M3, M4]). 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 the slave, warnings would be sent out (However, this is not shown in the diagrams).

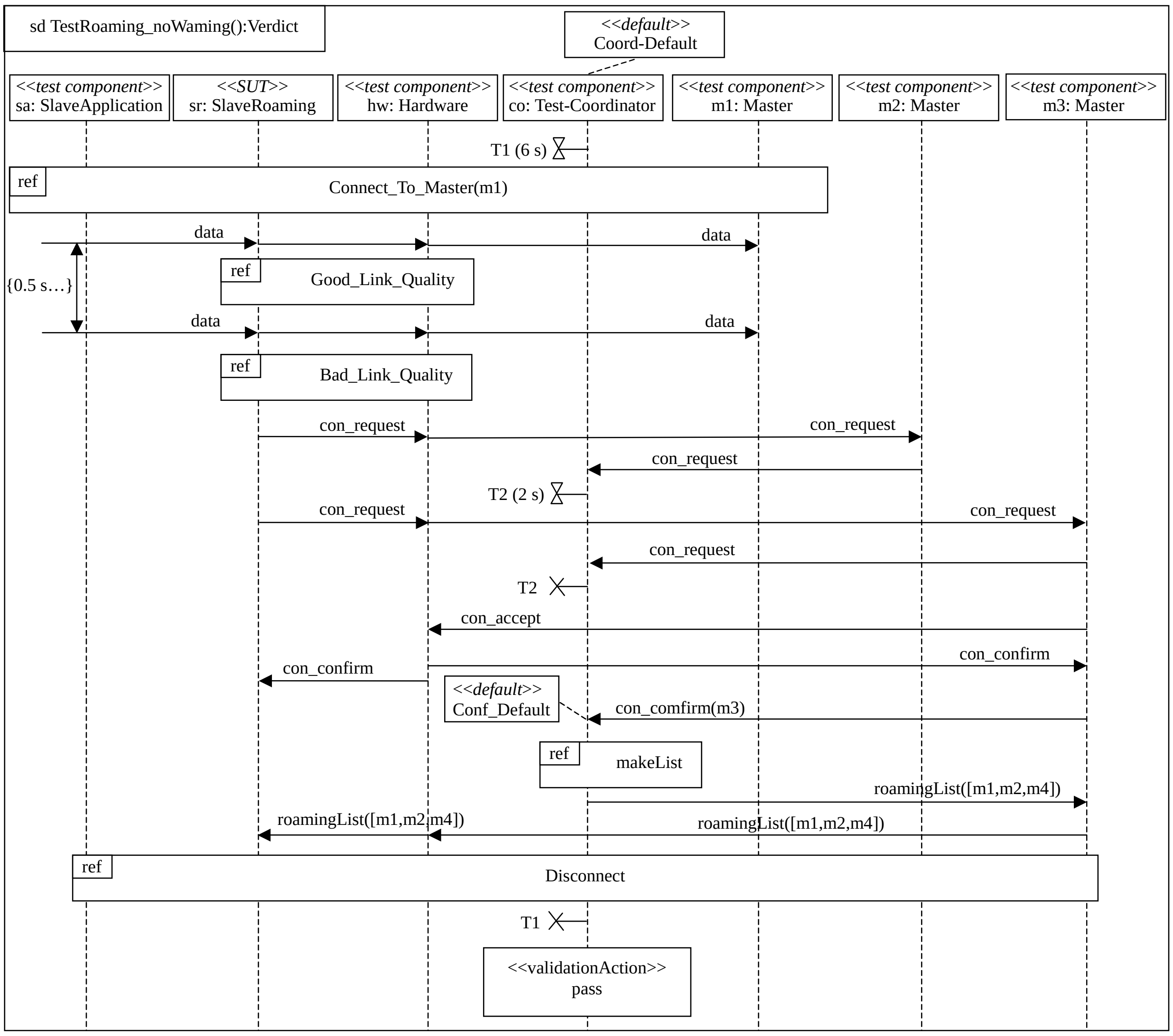
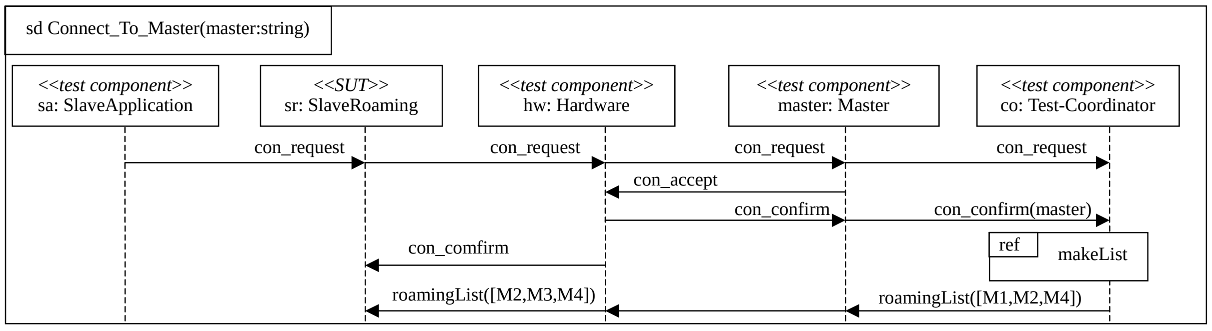


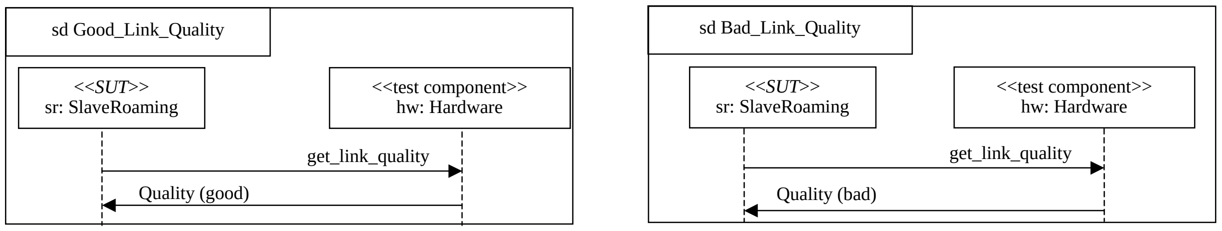
Fig.4 Test scenario

When the referenced behavior of Connect_to_Master has finished in Fig.4, the slave has successfully connected to master m1 and SlaveApplication starts to send data to the master. Additionally, the link quality is checked periodically. The time constraint of 0.5 seconds is specified to assure the function Good_Link_Quality, which is performed every 0.5 seconds, is executed before sending the second data package. Checking the link quality is specified in the functions Good_Link_Quality and Bad_Link_Quality in Fig. 5b. Herein, SlaveRoaming triggers the evaluation request and receives the result from the hardware.

In the first part of test case TestRoaming_noWarning (Fig.4), the Hardware has to be tuned to report a good link quality. Thus, further data can be sent. In the second part, the link quality is determined to be bad. Therefore, a new master is looked up. According to the roaming list, the new master must be m2. A connection request is expected to be sent to m2 by the SUT. As soon as it is observed and reported to the Test-Coordinator, a timer T2 of two seconds is started. This timer assures that when the SUT cannot establish a connection to a master, the SUT chooses a further master and tries to connect to it within two seconds.


sd Connect_To_Master(master:string)
<<test component>> sa: SlaveApplication
<<SUT>> sr: SlaveRoaming
<<test component>> hw: Hardware
<<test component>> master: Master
<<test component>> co: Test-Coordinator
con_request
con_request
con_request
con_request
con_accept
con_confirm
con_confirm(master)
ref makeList
con_comfirm
roamingList([M2,M3,M4])
roamingList([M1,M2,M4])

(a)



(b)

Fig.5 Test Functions (a) Connect to master function; (b) Link quality evaluation functions

If it is observed that the SUT requests a connection to the correct master m3, the timer T2 is stopped by the Test-Coordinator. In this test case, the connection is accepted (con_accept) by master m3 and hence confirmed (con_confirm). After the Test-Coordinator noticed the connection to the correct master, it assembles the new roaming list and sends it via the master to the slave. In case that no connection confirmation is received, the default behavior Conf_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.

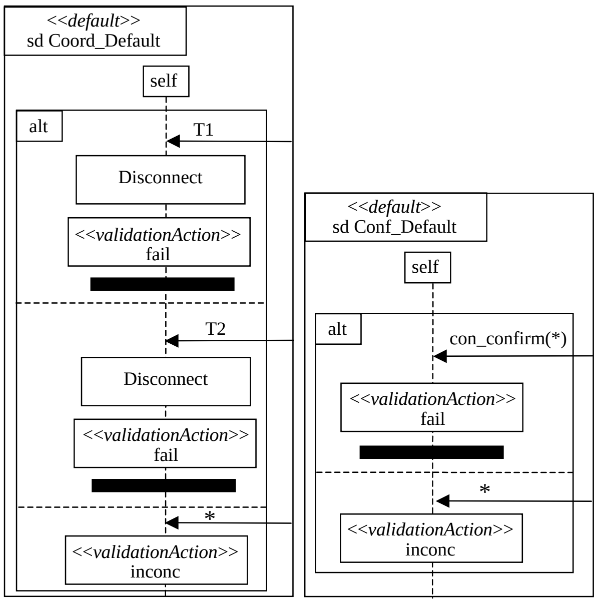
Besides the expected test behavior of test case TestRoaming_NoWarning, default behaviors are specified to catch the observations which lead to a fail or inconclusive verdict. The given test case uses two defaults called Coord_Default and Conf_Default (Fig.6). In U2TP, test behaviors can be specified by all UML behavioral diagrams, including interaction diagrams, state machines and activity diagrams. Thus, Fig.6 shows how default behaviors can be specified either as sequence diagrams (Fig.6a) or as state machines (Fig.6b).

Coord_Default is an instance-specific default applied to the coordinator. It defines three alternatives. The first two alternatives catch the timeout events of the timers T1 and T2. In both cases, slave and master will be disconnected and the verdict is set to fail. After that, the test component terminates itself. The third alternative catches any other unexpected events. In this case, the verdict is set to inconclusive and the test behavior returns back to the test event which triggered the default.

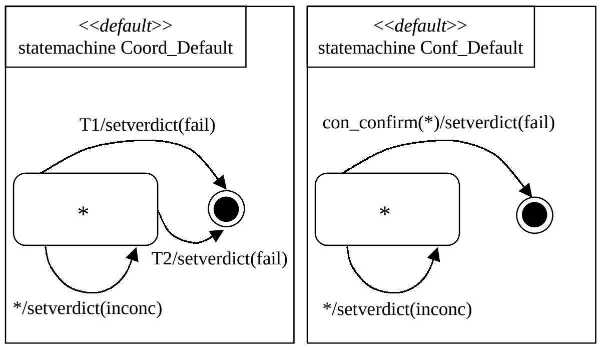
Conf_Default is an event-specific default attached to the connection confirmation event. In the Test-Coordinator, this default is invoked if either the connection confirmation is not sent from the correct master or another message than the connection confirmation is received. In the first case, the verdict 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.

CONCLUSION AND OUTLOOK

In this paper, we have presented a case study of how to use the newly adopted U2TP, in which some



(a)



(b)

Fig.6 Test defaults (a) Default as sequence diagrams; (b) Default as statemachines

of the authors were involved. The U2TP is a UML profile which allows the specification of black-box tests based on the new version 2.0 of UML.

We proposed a methodology for deriving test models from existing design model and demonstrated in the case study its applicability by developing a test context for a Bluetooth roaming model.

Further study is required to investigate transformation rules and automatic derivation of test models from design model. Additionally, it would be interesting to assess the possibility of hardware test specification using the U2TP.

References

Beizer, B., 1995. Black-Box Testing. John Wiley & Sons, Inc.

Dai, Z.R., Grabowski, J., Neukirchen, H., Pals, H., 2004. From Design to Test−Applied to a Roaming Algorithm for Bluetooth Devices. Next Generation Testing for Next Generation Networks. Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems (TestCom 2004), LNCS 2978, Springer, Oxford, United Kingdom.

Eriksson, H.E., Penker, M., Lyons, B., Fado, D., 2003. UML 2 Toolkit. Wiley Publisher, ISBN: 0471463612.

ISO/IEC, 1994. Information Technology-OSI–Conformance Testing Methodology and Framework. International ISO/IEC multi-part standard No. 9646.

Pals, H., Dai, Z.R., Grabowski, J., Neukirchen, H., 2003. UML-Based Modelling of Roaming with Bluetooth Devices. First Hangzhou-Luebeck Conference on Software Engineering, HL-SE'03.