Tracing Requirements In A Model-Based Testing Approach

Fredrik Abbors, Dragoş Truşcan, and Johan Lilius,
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5 A, 20520, Turku, Finland
Email: {Fredrik.Abbors, Dragos.Truscan, Johan.Lilius}@abo.fi

Abstract

In this paper we discuss an approach for requirements traceability in a model-based testing process. We show how the informal requirements of the system under test evolve and are traced at different steps of the process. More specifically, we discuss how requirements are traced to system specifications and from system specification to tests during the test generation process, and then how the test results are analyzed and traced back the specification of the system. The approach allows us to have both a fast feed-back loop for debugging either the specification or the implementation of the system and a way to estimate the coverage degree of the generated tests with respect to requirements. We discuss tool support for the approach and exemplify with excerpts from a case study in the telecommunications domain.

1. Introduction

The key to successful product engineering in the software industry today is in many cases a good quality-assurance and deployment of software systems. Customers demand highly efficient, low-cost, and reliable software products. Companies are forced to build high-end products with a low budget in a short time. The increasing demand for software products forces the companies to develop new products at a very fast pace. We see a constant decrease in the time-to-market and customers demanding more flexible systems which result in the growing system complexity. Missing the deadline for the time-to-market can have a huge negative impact on the company's profit. Unfortunately, this fast pace leaves the companies with less time for testing their products.

The purpose of testing is to find faults that have been introduced during the development of the system, starting with the initial specification phases and ending with its implementation. Testing is also a means to ensure the quality of a product and to verify that the product meets its requirements. Having a systematic and automated way to test software systems would reduce the overall expenses of testing due to a shorter time-to-market. Likewise, this would leave the companies with more time for actually designing and implementing the software, and would result in better and more reliable software.

Model-based testing (MBT) is a software testing technique that has gained much interest in recent years by providing the degree of automation needed for shortening the time required for testing. The main idea behind MBT, is that a behavioral model of the system, namely a *test model*, is used for automatically deriving *test cases* following different coverage criteria.

The test model is typically developed from the informal requirements of the system and therefore it is important to trace how the generated test cases cover different requirements. Traceability of requirements can help one to achieve the right level of coverage and show what requirement has been covered by what test. Traceability can also be used to trace requirements to specifications (code or models) and can detect what part of a code or a model implements a requirement. By tracing requirements to tests, it becomes possible to trace back requirements to models, when a test fails and to identify from which part of a the test model the failure originated.

In this paper we present an approach for tracing product requirements across a model-based testing process, from informal documents via test models to test cases, and back to requirements and test models. The approach allows us to have both a fast feedback loop for debugging either the specification or the implementation of the system and a way to estimate the coverage degree of the generated tests with respect to requirements. We discuss tool support for the approach and exemplify with excerpts from a case study in the telecommunications domain.

Related Work. Requirements traceability is a very popular topic in the software engineering and testing communities, and has gained momentum in the context of model-based testing in the context of automated test generation. However, as requirements change during the development life cycles of software systems, updating and managing traces has become a tedious task. Researchers have addressed this problem by developing methods for automatic generations of traceability relations [1] [2] [3] [4] by using information retrieval techniques to link artifacts together based on common key-words that occur in both the requirement description and in a set of searchable documents. Other approaches focus on annotating the model with requirements which are propagated through the test generation process in order to obtain a requirement traceability matrix [5]. The matrix is then used to manually analyze and track requirements to models. From the reviewed works, the one in [6] is closer to our approach. In there, the authors use textual delimiters to add requirements in the OCL constructs associated to a restricted set of UML models. The LEIROS test design tool is then used to generate test cases, and a traceability matrix is obtained after the test are executed. However, there is no tool support for tracing-back requirements from tests to models.

2. Model-Based Testing Process

Our model-based testing process (Figure 1) starts with the analysis and structuring of the informal requirements into a *Requirements Model*. The Requirements Diagrams of the Systems Modeling Language (SysML) [7] are used for this purpose. In the next phase, the system under test (SUT) is specified using the Unified Modeling Language (UML) [8]. In our modeling process, we consider that several perspectives of the SUT are required in order to enable a successful test derivation process later. In addition, one should note that in our approach a model of the system is used for deriving test cases and not a test model, the difference between the two being that the former is both used for development and testing, whereas the latter is only used for testing. Several perspectives of the SUT are modeled; a class diagram is used to specify a *domain model* showing what domain components exist and how they are interrelated through interfaces. A *behavioral model* describes the behavior of the SUT using state machines. *Data models* are used to describe the message types exchanged between different domain entities. Last but not least, *domain configuration* models are used to represent specific test configurations using object diagrams.

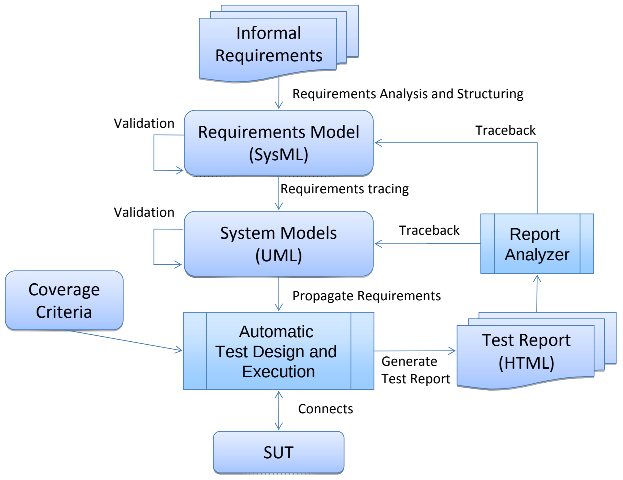


Figure 1: Overview of the model-based testing process

A set of modeling guidelines and validation rules are defined for ensuring the quality of the resulting models. Modeling guidelines are used to specify how different models are created from requirements or from other models, what information they should contain, how this information is related to the information present in the other models, etc.

Validation rules have been defined and implemented [9] for both Requirements Models and for System Models for checking different quality metrics of the resulting models before proceeding to the *test derivation phase*. These rules ensure that the models are syntactically correct, they are consistent with each other, and that they contain the information needed in the later phases of the testing process. Tool support is provided for automatically verifying these rules using the Object Constraint Language (OCL). The OCL rules check the static semantics of the models and can be used to describe constraints that are specific to the domain, modeling language, modeling process, etc. However, if OCL can be used for checking the dynamic semantics of the models has to be further investigated. The NoMagic's MagicDraw tool [10] has been used for editing the SysML and UML models and for running the validation rules.

The models used to specify the SUT are subsequently transformed into input for an automated test derivation tool, namely Conformiq's Qtronic [11]. We use the *online testing* mode of this tool, in which tests are generated and applied on-the-fly against the SUT. The desired coverage criteria used for test generation are manually selected from the graphical user interface (GUI) of Qtronic. At the end of each test run, a automatically generated *Test Report* will summarize the result of the testing process in terms of generated test cases, verdicts, coverage levels, requirements traceability matrix, etc.

3. Requirements Traceability

Our approach to requirements traceability is built on top of the previously explained testing process with two goals in mind. Firstly, we want to be able to trace how different parts of the system models relate to the requirements and then to see how different requirements are covered by the generated test cases. Another reason for tracing requirements is that if a requirement changes, it is essential to know how this change is reflected in the models [12] [13]. Secondly, once the test report becomes available, we would like to be able to identify which requirements have been successfully tested and which have resulted in failures. In addition, for the failed test cases we should be able to trace back from test cases those parts of the SUT specification that generated the failure.

In the following, we briefly describe our requirements traceability approach while providing small examples from a telecommunications case study. In our case, the SUT is a Mobile Switching Server (MSS). The MSS is a network element located in a mobile telecommunication system. The MSS is connected to its surrounding elements through several different interfaces. The MSS is responsible for keeping track of the location of mobile subscribes (MS) in the network and for connecting calls between MS's over 2G and 3G networks. The MSS is also responsible for tracking the movement of MS's during an ongoing call.

3.1. Tracing requirements to tests

3.1.1. Requirements decomposition. The requirements models are obtained by analyzing informal requirements related to standards, protocols, system specifications, etc. Requirements are structured in a tree-like manner and defined on several levels of abstraction following a functional decomposition. They can also be related (i.e. traced) to other requirements on the same abstraction level. Requirements may also be decomposed into different categories, depending on the nature of the requirement, like functional, architectural, data, etc.

Each requirement element contains a *name* field which specifies the name of the requirement, an *id* field, and a *text* field. The *id* field simply specifies the id of the requirement, whereas the text field describes the requirement. A requirement also contains a *source* field. The source field specifies the origins of the

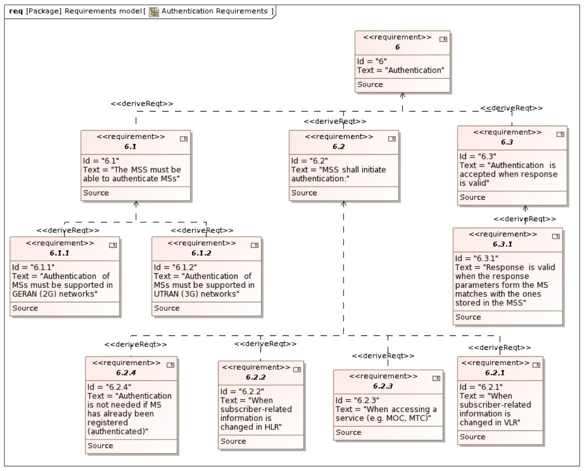
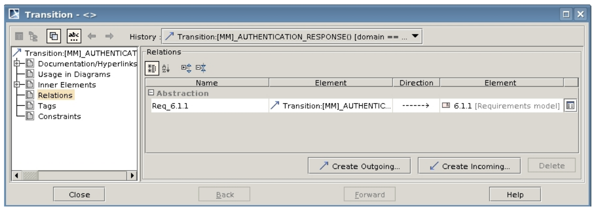

req [Package] Requirements model[Authentication Requirements]
<<requirement>> 6
Id = "6"
Text = "Authentication"
Source
<<deriveReqt>>
<<requirement>> 6.1
Id = "6.1"
Text = "The MSS must be able to authenticate MSs"
Source
<<deriveReqt>>
<<requirement>> 6.2
Id = "6.2"
Text = "MSS shall initiate authentication:"
Source
<<deriveReqt>>
<<requirement>> 6.3
Id = "6.3"
Text = "Authentication is accepted when response is valid"
Source
<<deriveReqt>>
<<requirement>> 6.1.1
Id = "6.1.1"
Text = "Authentication of MSs must be supported in GERAN (2G) networks"
Source
<<deriveReqt>>
<<requirement>> 6.1.2
Id = "6.1.2"
Text = "Authentication of MSs must be supported in UTRAN (3G) networks"
Source
<<deriveReqt>>
<<requirement>> 6.3.1
Id = "6.3.1"
Text = "Response is valid when the response parameters form the MS matches with the ones stored in the MSS"
Source
<<deriveReqt>>
<<requirement>> 6.2.4
Id = "6.2.4"
Text = "Authentication is not needed if MS has already been registered (authenticated)"
Source
<<deriveReqt>>
<<requirement>> 6.2.2
Id = "6.2.2"
Text = "When subscriber-related information is changed in HLR"
Source
<<deriveReqt>>
<<requirement>> 6.2.3
Id = "6.2.3"
Text = "When accessing a service (e.g. MOC, MTC)"
Source
<<deriveReqt>>
<<requirement>> 6.2.1
Id = "6.2.1"
Text = "When subscriber-related information is changed in VLR"
Source

Figure 2: Example of a SysML requirements diagram



Name	Element	Direction	Element
Abstraction			
Req_6.1.1	Transition:[MM]_AUTHENTIC...	------>	6.1.1 [Requirements model]

Figure 3: Linking requirements to a transition in a state machine

requirement. The source can be a link to or a name of a textual document from where the requirement has been extracted. Figure 2 shows the functional requirements for the location update procedure of the MSS represented using a SysML requirements diagram.

3.1.2. Requirements traced to models. The UML models of the SUT are built starting from the requirements models. During this process, the requirements are traced to different parts of the models to point how each requirement is addressed by the models. The relationships between requirements and models are specified on several levels. Non-leaf requirements are refined (linked) to models, e.g. state machine models. An exceptional situation is in the case of the top-level functional requirements, which are linked to use cases in the use case model of the SUT. The leaf requirements in the requirements tree are then linked to other UML elements to which they apply, e.g. transitions in a state machine or classes in a class diagram. Figure 3 shows how a requirement can be linked to a model element, e.g. a transition in a state machine using the MagicDraw editor.

This is done to ensure the traceability of requirements within the system models and to test cases.

These links are useful for evaluating (using the previously discussed validation rules) whether all the requirements have been reflected in the models or by showing what elements from different diagrams specify a given requirement. When all requirements have been linked to model elements and the models have been validated, the UML models are transformed into input for the Qtronic tool via an automated transformation.

The transformation [14] [15] basically translates UML models to the Qtronic Modeling Language (QML), the language used by Qtronic for specifying the SUT. QML is a textual specification language with a Java-like syntax in which one can specify the input/output ports of the system and what data types (complex data type are supported) can be send and received on different ports. The behavior of the SUT can be described either in QML or using a simplified version of UML statecharts. In the latter case, QML can be used as an action language for the statechart.

Qtronic provides support for requirement coverage during test generation. Requirements are associated to state models, more precisely to the actions on transitions via the `requirement` statement. Basically, the requirements in Qtronic are tags that are used to trace if a specific transition in the state model has been covered by the generated test cases.

During the transformation from UML to QML, links between requirements and model elements are preserved. In the current status of our work, only requirements attached to state machine transitions are propagated to Qtronic. Requirement hierarchy is specified in QML with the "/" character. Figure 4 shows an example of a state machine that has been transformed from UML to QML. In this figure, one can see that `requirement 6.1.1` and `requirement 6.1.2` in the MagicDraw model are propagated to the same transitions in QML. As the rest of the system description in QML is not relevant for this paper we do not include it here. However a detailed example can be found in [15].

3.1.3. Tracing requirements to tests.

In Qtronic, test cases are generated according to different coverage criteria, like requirements coverage, transition coverage, state coverage. The coverage criteria are selected manually using the Qtronic user interface. By combining one or more of the mentioned criteria, Qtronic tries to generate tests based on those criteria. If the requirements coverage criterium is enabled, one can choose to test different requirements individually, by checking or unchecking the corresponding requirement in a list. Qtronic will then generate test cases that cover

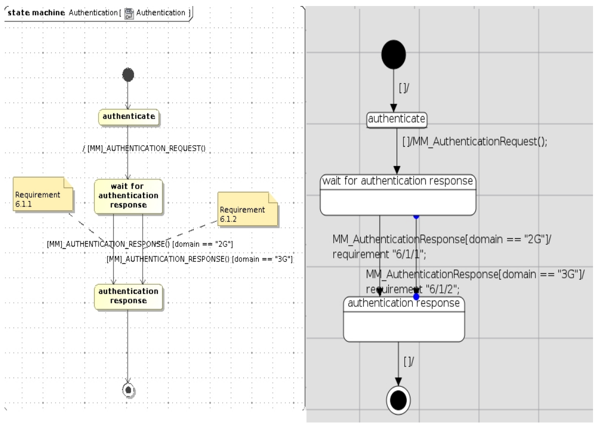

state machine Authentication [Authentication]
authenticate
/ [MM]_AUTHENTICATION_REQUEST()
Requirement 6.1.1
wait for authentication response
Requirement 6.1.2
[MM]_AUTHENTICATION_RESPONSE() [domain == "2G"]
[MM]_AUTHENTICATION_RESPONSE() [domain == "3G"]
authentication response
[]/
authenticate
[]/MM_AuthenticationRequest();
wait for authentication response
MM_AuthenticationResponse[domain == "2G"]/ requirement "6/1/1";
MM_AuthenticationResponse[domain == "3G"]/ requirement "6/1/2";
authentication response
[]/

Figure 4: Example of a UML state machine in MagicDraw (left) and its equivalent in QML (right)

the selected requirements.

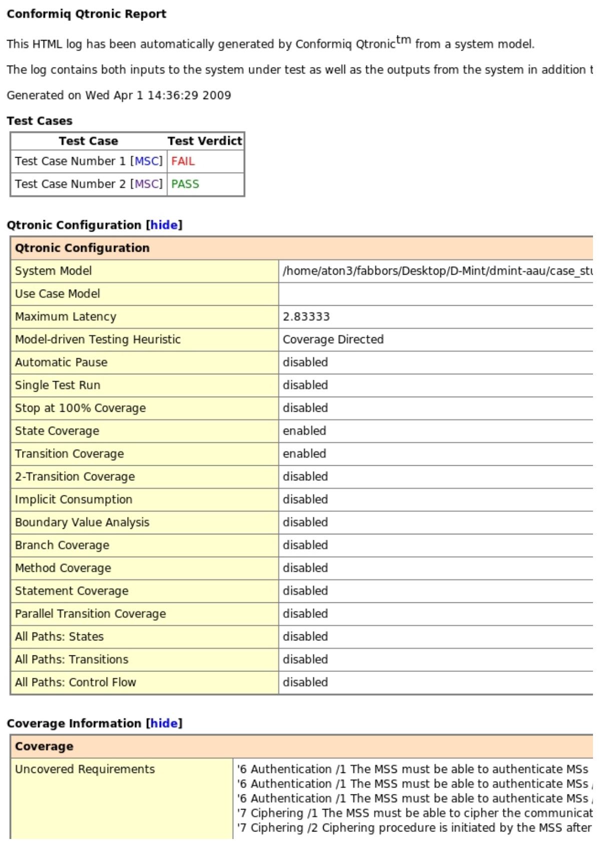
In the online testing mode, Qtronic handles the *test execution* process. One-by-one it generates an input message, sends it via the adapter to the SUT, and generates a new input message based on the responses from the SUT. A logging back-end can be used during test execution. The logging back-end provides connectivity to the Qtronic reporting infrastructure and it is used by Qtronic to generate a test report. Three logging back-ends are provided by default. With these logging back-ends, Qtronic can generate test reports in HTML, SQLite, and XML format. When all tests have been applied against the SUT, Qtronic generates a test report in the chosen format. Listing 1 shows an example of a generated test case specified in XML. As one can notice, the requirements have been propagated during test generation and included in the test specification (see line 6).

Listing 1: Requirement propagated to Qtronic test specification

```
1  <checkpoint>
2   <symbol value="transition: LocationUpdate-
        Authentication->LocationUpdate-Ciphering-
        initial-0-1"/>
3   <timestamp nanoseconds="447362000" seconds="0"/>
4  </checkpoint>
5  <checkpoint>
6   <symbol value="6 Authentication /1 The MSS must
        be able to authenticate MSs /1
        Authentication of MSs must be supported in
        GERAN (2G) networks"/>
7   <timestamp nanoseconds="447399000" seconds="0"/>
8  </checkpoint>
9  <checkpoint>
```

Figure 5 shows a example of a test report[1] generated

1. The test report also includes a requirements traceability matrix which we do not include due to space reasons.



Conformiq Qtronic Report

This HTML log has been automatically generated by Conformiq Qtronic^{tm} from a system model.

The log contains both inputs to the system under test as well as the outputs from the system in addition t

Generated on Wed Apr 1 14:36:29 2009

Test Cases

Test Case	Test Verdict
Test Case Number 1 [MSC]	FAIL
Test Case Number 2 [MSC]	PASS

Qtronic Configuration [hide]

Qtronic Configuration	
System Model	/home/aton3/fabbors/Desktop/D-Mint/dmint-aau/case_stu
Use Case Model	
Maximum Latency	2.83333
Model-driven Testing Heuristic	Coverage Directed
Automatic Pause	disabled
Single Test Run	disabled
Stop at 100% Coverage	disabled
State Coverage	enabled
Transition Coverage	enabled
2-Transition Coverage	disabled
Implicit Consumption	disabled
Boundary Value Analysis	disabled
Branch Coverage	disabled
Method Coverage	disabled
Statement Coverage	disabled
Parallel Transition Coverage	disabled
All Paths: States	disabled
All Paths: Transitions	disabled
All Paths: Control Flow	disabled

Coverage Information [hide]

Coverage	
Uncovered Requirements	'6 Authentication /1 The MSS must be able to authenticate MSs '6 Authentication /1 The MSS must be able to authenticate MSs , '6 Authentication /1 The MSS must be able to authenticate MSs , '7 Ciphering /1 The MSS must be able to cipher the communicat '7 Ciphering /2 Ciphering procedure is initiated by the MSS after

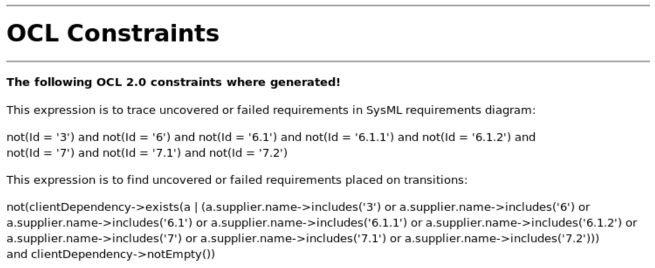
Figure 5: Test report produced by Qtronic

by Qtronic with a HTML logging back-end. It is also possible to inspect each test case individually by clicking on the *[MSC]* link next to the test case number in the test report. This will bring up a message sequence chart (MSC) showing the order of messages sent and received by Qtronic.

3.2. Back-tracing of requirements

The approach opposite to the one presented above, is to trace-back requirements from test cases to models. For this purpose, we analyze the test report, collect the information of the failed test cases, and trace the requirements attached to those test cases, back to system models. This way we can see which requirements were not validated during testing and to what parts of the specification they are linked.

We have developed a Python script that automatically analyzes the Qtronic test report and generates a set of OCL queries (see Figure 6), that we use in MagicDraw to locate erroneous parts in the UML system models. In this way, we can see which requirements failed during testing and to what model elements they are linked. Figure 7 shows how requirements, which



OCL Constraints

The following OCL 2.0 constraints where generated!

This expression is to trace uncovered or failed requirements in SysML requirements diagram:

```
not(Id = '3') and not(Id = '6') and not(Id = '6.1') and not(Id = '6.1.1') and not(Id = '6.1.2') and
not(Id = '7') and not(Id = '7.1') and not(Id = '7.2')
```

This expression is to find uncovered or failed requirements placed on transitions:

```
not(clientDependency->exists(a | (a.supplier.name->includes('3') or a.supplier.name->includes('6') or
a.supplier.name->includes('6.1') or a.supplier.name->includes('6.1.1') or a.supplier.name->includes('6.1.2') or
a.supplier.name->includes('7') or a.supplier.name->includes('7.1') or a.supplier.name->includes('7.2')))
and clientDependency->notEmpty())
```

Figure 6: OCL constraints produced by the Req2Ocl script

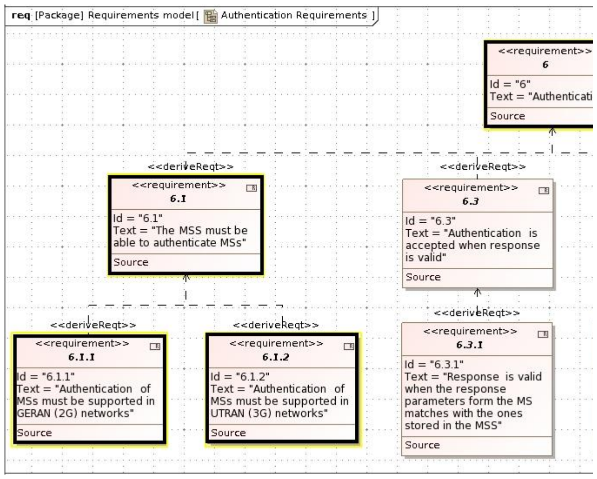

req [Package] Requirements model [Authentication Requirements]
<<requirement>>
6
Id = "6"
Text = "Authenticati
Source
<<deriveReqt>>
<<requirement>>
6.1
Id = "6.1"
Text = "The MSS must be able to authenticate MSs"
Source
<<deriveReqt>>
<<requirement>>
6.3
Id = "6.3"
Text = "Authentication is accepted when response is valid"
Source
<<deriveReqt>>
<<requirement>>
6.1.1
Id = "6.1.1"
Text = "Authentication of MSs must be supported in GERAN (2G) networks"
Source
<<deriveReqt>>
<<requirement>>
6.1.2
Id = "6.1.2"
Text = "Authentication of MSs must be supported in UTRAN (3G) networks"
Source
<<deriveReqt>>
<<requirement>>
6.3.1
Id = "6.3.1"
Text = "Response is valid when the response parameters form the MS matches with the ones stored in the MSS"
Source

Figure 7: Tracing of requirements to a SysML Requirement Diagram

failed during testing, are found in the requirements model with the help of the OCL queries. In Figure 8 one can see how the same requirements are found in the state machine diagram, on the same transitions to which they were initially traced. Ultimately, since requirements are traced to model elements, it facilitates the identification of which functionalities of SUT are not in sync with the *model*, and hence with the requirements.

4. Conclusion

This paper has presented an approach for traceability of requirements in a model-based testing approach. We have shown how requirements can be traced to models, to test specifications, and back to models again. Traceability of requirements facilitates the process of locating parts in the system models that are causing failed test cases. Further, traceability of requirements can help in depicting missing tests, i.e. when critical requirements are not traced to any tests.

Currently, our presented approach only supports

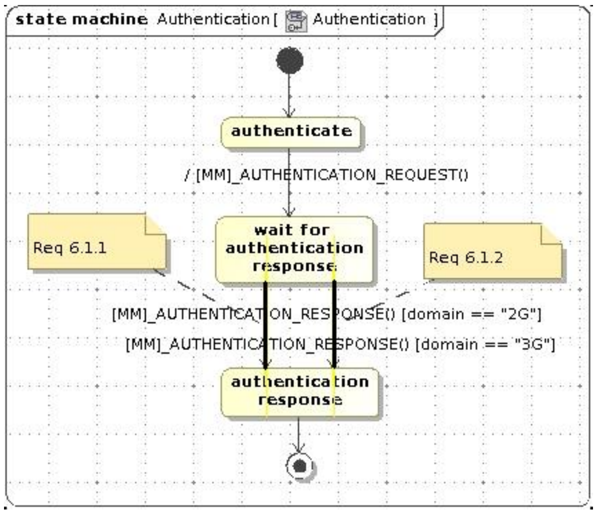

state machine Authentication [Authentication]
authenticate
/ [MM]_AUTHENTICATION_REQUEST()
wait for authentication response
Req 6.1.1
Req 6.1.2
[MM]_AUTHENTICATION_RESPONSE() [domain == "2G"]
[MM]_AUTHENTICATION_RESPONSE() [domain == "3G"]
authentication response

Figure 8: Tracing of requirements to transitions in a state machine

online testing. In the future, we will extend it to offline testing, as well. Another future goal is to find a solution for tracing non-functional requirements to system models and, respectively, to tests. This is something that has not yet been fully investigated.

Our approach benefits from a well integrated tool chain, in which specialized tools are used for each phase of the model-based testing process. When not already provided by the tools involved, we have provided automation of the transitions between the phases of the process, allowing to have a fast feed-back loop for testing and debugging the specifications or the implementation of the SUT. In addition, having a fully automated approach, the effort in updating the models and performing the testing again was diminished.

The approach also proved beneficial through the fact that many errors have been detected in the early stages of the process, when the system models have been created. The errors were caused mainly by omissions in the models and by misinterpreting the requirements. Thus, when failed test cases were reported after test runs we could focus our attention directly on debugging the implementation of the SUT.

References

[1] G. Spanoudakis, A. Zisman, E. Perez-Minana, and P. Krause, "Rule-Based Generation of Requirements Traceability Relations," *The Journal of Systems & Software*, vol. 72, no. 2, pp. 105–127, 2004.

[2] C. Duan and J. Cleland-Huang, "Visualization and Analysis in Automated Trace Retrieval," in *Requirements Engineering Visualization, 2006. REV'06. First International Workshop on*, 2006, pp. 5–5.

[3] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou, "Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability," in *13th IEEE International Conference on Requirements Engineering, 2005. Proceedings*, pp. 135–144.

[4] J. Hayes, A. Dekhtyar, and J. Osborne, "Improving Requirements Tracing via Information Retrieval," in *11th IEEE International Requirements Engineering Conference, 2003. Proceedings*, 2003, pp. 138–147.

[5] F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting, "Requirements Traceability in Automated Test Generation: Application to Smart Card Software Validation," in *Proceedings of the 1st international workshop on Advances in model-based testing*. ACM New York, NY, USA, 2005, pp. 1–7.

[6] E. Bernard and B. Legeard, "Requirements Traceability in the Model-Based Testing Process," in *Software Engineering*, ser. Lecture Notes in Informatics, vol. 106. Bttinger, Stefan and Theuvsen, Ludwig and Rank, Susanne and Morgenstern, Marlies, 2007, pp. 45–54.

[7] Object Management Group, "OMG SysML Specification," Tech. Rep. [Online]. Available: http://www.omg.org/spec/SysML/1.1/

[8] "Unified Modeling Language - http://www.omg.org/spec/UML/2.0/." [Online]. Available: http://www.omg.org/spec/UML/2.0/

[9] J. Abbors, "Increasing Quality of UML Models Used for Automatic Test Generation," Master's thesis, bo Akademi University, 2009.

[10] "NoMagic MagicDraw," http://www.magicdraw.com/.

[11] "Conformiq Qtronic," http://www.conformiq.com/.

[12] T. Tsumaki and Y. Morisawa, "A Framework of Requirements Tracing using UML," in *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, 2000, pp. 206–213.

[13] R. Settimi, J. Cleland-Huang, O. Khadra, J. Mody, W. Lukasik, and C. DePalma, "Supporting software evolution through dynamically retrieving traces to UML artifacts."

[14] T. Pääjärvi, "Generation Input for the Test Generator Tool from UML Design Models," Master's thesis, Åbo Akademi University, 2009.

[15] F. Abbors, T. Pääjärvi, R. Teittinen, D. Truşcan, and J. Lilius, "A Semantic Transformation from UML Models to Input for the Qtronic Test Design Tool," Turku Centre for Computer Science (TUCS), Tech. Rep. 942, 2009.