	<b>Title:</b> Common Approach to Architecture Driven Testing in the D-MINT Project – White Paper
	<b>Version:</b> 1.6 <b>Date :</b> 6.11.2009 <b>Pages :</b> 51
	<b>Editors:</b> Fraunhofer FOKUS, Åbo Akademi
<p>The D-MINT Consortium consists of:          ABB, Åbo Akademi University, Conformiq Software, Daimler, DataPixel, ELIKO, Elvior, ETSI, Fraunhofer IESE, Fraunhofer FOKUS, IDEKO, Innovalia, INSPIRE, iXtronics, Nethawk, Nokia Siemens Networks, PIKETEC, Simula, SQS, SORALUCE, Tallinn University of Technology, TANDBERG, Testing Technologies, TRIMEK, VTT Technical Research Centre of Finland</p>	<b>To:</b> D-MINT Consortium
<b>Printed on:</b> 11/6/2009 2:55:00 PM	
<b>Status:</b> <input type="checkbox"/> Draft <input type="checkbox"/> To be reviewed <input type="checkbox"/> Proposal <input checked="" type="checkbox"/> Final / Released	<b>Confidentiality:</b> <input checked="" type="checkbox"/> Public - Intended for public use <input type="checkbox"/> Restricted - Intended for D-MINT consortium only <input type="checkbox"/> Confidential - Intended for individual partner only
<b>Deliverable ID:</b> CA_WP_v1.6	
<b>Title:</b> <b>Common Approach to Architecture-Driven Testing</b>	
<b>Summary / Contents:</b> <p>This document gives an overview of the common approach to architecture-driven testing pursued in the D-MINT project.</p>	


## TABLE OF CONTENTS

<b>1. Introduction.....</b>	<b>8</b>
<b>2. Architecture-Driven Model-based Testing Process in D-MINT .....</b>	<b>8</b>
2.1 Functional View .....	9
2.2 Logical View .....	10
2.3 Technical View .....	10
2.4 Topology View.....	10
<b>3. Architecture-Driven Approach as a Test Creation Methodology.....</b>	<b>11</b>
3.1 Requirements Management and Documentation.....	13
3.2 Modeling for Test Derivation .....	13
3.3 Test Derivation .....	14
3.4 Test Implementation.....	15
3.5 Test Execution .....	15
3.6 Test Reporting.....	15
<b>4. The Common Approach Based on the Selected Case Studies.....</b>	<b>15</b>
4.1 ABB Case Study .....	16
4.2 Daimler Case Study .....	18
4.3 Eliko Case Study .....	21
4.4 ETSI Case Study.....	22
4.5 IDEKO/Soraluce Case Study .....	23
4.6 NSN Case Study .....	25
4.7 Trimek Case Study.....	26
<b>5. Annex.....</b>	<b>27</b>
5.1 Requirements Management and Documentation.....	27
5.1.1 Methods .....	27
5.1.2 Notations.....	27
5.1.3 Tools .....	29
5.2 Modeling for Test Derivation .....	29
5.2.1 From requirements to System Model.....	30
5.2.1.1 Methods .....	30
5.2.1.2 Notations .....	32
5.2.1.3 Tools .....	33
5.2.2 From requirements to Test Model.....	34
5.2.2.1 Methods .....	34
5.2.2.2 Notations .....	36
5.2.2.3 Tools .....	37
5.3 Test Derivation .....	38
5.3.1 Test Derivation From System Models.....	38
5.3.1.1 Methods .....	38
5.3.1.2 Notations .....	39
5.3.1.3 Tools .....	39
5.3.2 Test Derivation from Test Models.....	41
5.3.2.1 Methods .....	41
5.3.2.2 Notations .....	42
5.3.2.3 Tools .....	43
5.4 Test Implementation.....	43

5.4.1	Methods .....	44
5.4.2	Notations.....	44
	Tools .....	45
5.4.3	.....	45
5.5	Test Execution .....	46
5.5.1	Methods .....	46
5.5.2	Notations.....	47
5.5.3	Tools .....	47
5.6	Test Reporting.....	50
5.6.1	Methods .....	50
5.6.2	Notations.....	50
5.6.3	Tools.....	50

## CHANGE LOG

Vers.	Date	Author	Description
0.1	10.04.09	George Din, K.-D. Engel	Creation, TOC
0.1	11.05.09	Justyna Zander	TOC enhancement, Layout
0.1	12.05.09	Justyna Zander	Merging with the former version of the TOC.
0.1	13.05.09	Justyna Zander	Introduction, Case Studies
0.2	18.05.09	Andres Kull	Eliko CS
0.2	04.06.09	Justyna Zander	Tables update
0.2	10.06.09	Justyna Zander	ETSI CS
0.2	18.06.09	Justyna Zander	Part 2 – theory
0.3	19.06.09	Dragos Truscan	Part 2 – theory, added by Justyna
0.4	25.6.09	K.-D.Engel	Added Daimler/IESE Case Study
0.5	7.7.09	Kde	Added individual tables to case studies Moved Common table from 2.2 -> 2.1 (removed 2.2) Included change requests by Ideko Included change requests by ABB
0.6 (14)	16.07.09	Dragos Truscan	Typo fixing, text formatting Added more detailed explanation to section 2.2 Moved section 2.2.1 into 2.3
0.7	21.07.08	Kde	Revised vers. 14DT and included addition on StarUML, RSA, UML Seq charts, UTP; New tables from ABB added
0.8 (15)	29.07.08	Dragos Truscan	Added description for CamelView, Elvior's Motes, Elvior's MessageMagic, TTModeler, TT Workbench, Piketec TPT, EAST Additions from Fokus from v14-DT-KDE- revisedadditions_included, v07
0.8 (16)	17.08.09	Axel Rennoch	quality, prioritization, TM->ATS derivation, test implementation/execution
0.9 (17)	23.08.09	Dragos Truscan	Integrated comments from <a href="#">CA Paper D-MINT v15-DT--kde.doc</a> Inserted table and missing content from <a href="#">CaseStudies-CA-Info--15 Actual Tables.ppt</a> Added hyperlinks to table cells (Fig1) Additions made by Axel in <a href="#">CA Paper D-Mint V16.doc</a>
1.0 (19) (review)	31.08.09	Dragos Truscan	Description for Qtronic Modeler, QSEC (OPTIMIZE), JumBL Restructuring of section suggested (accepted by Klaus) Content from Alain (v18_restructured_alain.doc) on PTML Comments from Klaus and Axel Ordered references alphabetically Fixed missing text each and there.
1.1	25.09.09	Dragos Truscan	Moved content of sections 3.2-3.7 to Annex 6.1-6.6 Added update from Tomi on section 1.1 and 1.2 Added description of RootCause analysis in Test Reporting


	White Paper: V 1.6	Page : 5 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

			Updated D-Mint partner list (taken from the PCA) Added comments and JUMBL/JSeq tool descriptions from TBauer in CA_Paper_D-MINT_v1.0_IESE-tools.doc
1.2	01.10.09	Axel Rennoch	Sections 3.1.3 – 3.1.6
	01.10.09	Alain Vouffo	Added section on lessons learned based on internal survey results.
1.3	06.10.09	Dragos Truscan	Fixed typos in MessageMagicDescription Added description to 3.1.1 and 3.1.2 Fixed some broken references
1.4	15.10.09	Dragos Truscan	Removed section 6. Summary and lessons learned Refined Sec 3 Changed executive summary
1.5	16.10.09	Thomas Bauer	Chapter 1 rewritten
1.6	06.11.09	Dragos Truscan	Final version

## APPLICABLE DOCUMENT LIST

Ref.	Title, author, source, date, status
[AbbF09]	Fredrik Abbors. An Approach for Tracing Functional Requirements in Model-Based Testing; Master's Thesis; Åbo Akademi University; 2009.
[AbbJ09]	Johan Abbors. Increasing the Quality of UML Models Used for Automated Test Generation; Master's Thesis; Åbo Akademi University; 2009.
[ATL09]	Fredrik Abbors, Dragos Truscan, Johan Lilius. <a href="#">Tracing Requirements In A Model-Based Testing Approach</a> . The First International Conference on Advances in System Testing and Validation Lifecycle (VALID 2009)
[AV01]	Alain-Georges Vouffo Feudjio. Model-driven functional test engineering for service centric systems. In Proceedings of The 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom 2009). IEEE, 2009.
[B03]	Burnstein, I.: Practical Software Testing, Springer-Verlag, 2003.
[BAU09]	Thomas Bauer, Robert Eschbach, Martin Größl, Tanvir Hussain, Detlef Streitferdt, Florian Kantz. Combining combinatorial and model-based test approaches for highly configurable safety-critical systems. Proceedings of the Model-based Testing in Practice workshop (MOTIP 2009).
[Qtro09]	Conformiq Qtronic2 User Manual, version 2.0.2; Conformiq Software Ltd; January 16th, 2009.
[Conf]	Conformiq Qtronic product web page; <a href="http://www.conformiq.com/qtronic.php">http://www.conformiq.com/qtronic.php</a>
[OCL]	OMG Object Constraint Language (OCL); <a href="http://www.omg.org/spec/OCL/2.0/">http://www.omg.org/spec/OCL/2.0/</a>
[DAI06]	Dai, Z. R.: /An Approach to Model-Driven Testing with UML 2.0, U2TP and TTCN- 3. PhD Thesis, Technical University Berlin, ISBN: 978-3-8167-7237-8. Fraunhofer IRB Verlag, 2006.
[DM08]	D-MINT; D1.2.2, Nokia Siemens Networks Case Study and Initial Requirements; version 1.0; Thomas Deiß

[EA]	Enterprise Architect product web page; <a href="http://www.sparxsystems.com/products/ea/index.html">http://www.sparxsystems.com/products/ea/index.html</a>
[EM]	Elvior MOTES. <a href="http://www.elvior.ee/motes">http://www.elvior.ee/motes</a>
[EMM]	Elvior MessageMagic. <a href="http://www.elvior.ee/messagemagic">http://www.elvior.ee/messagemagic</a>
[ETSI07]	ETSI European Standard (ES) 201 873-1 V3.2.1 (2007-02): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute, Sophia-Antipolis, France. 2007.
[GPP]	3rd Generation Partnership Project web site; <a href="http://www.3gpp.org/">http://www.3gpp.org/</a>
[H09]	Andreas Hoffmann et. al.: A Generic Approach for Modeling Test Case Priorities with Applications for Test Development and Execution. MOTES'09 workshop at "Informatik 2009" Lübeck September 2009.
[JSEQ]	JUMBL resource website, University of Tennessee, USA <a href="http://sqr.eccs.utk.edu/esp/jumb1.html">http://sqr.eccs.utk.edu/esp/jumb1.html</a>
[JUMBL]	JSEQ resource website, Fraunhofer IESE, Department of Testing and Inspections <a href="http://tai.iese.fraunhofer.de">http://tai.iese.fraunhofer.de</a>
[KHJ07]	Kamga, J., Herrmann, J., Joshi, P.: Deliverable: D-MINT automotive case study - Daimler, Deliverable 1.1, Deployment of model-based technologies to industrial testing, ITEA2 Project, 2007.
[KTH05]	B. Korel, L. H. Tahat, and M. Harman, "Test Prioritization Using System Models" Proc. 21st IEEE Int'l Conf. Software Maintenance (ICSM '05), pp. 559-568, 2005.
[iXtronics]	iXtronics GmbH; <a href="http://www.ixtronics.com/">http://www.ixtronics.com/</a>
[LK08]	Lehmann, E., Krämer, A.: Model-based Testing of Automotive Systems. In Proceedings of IEEE ICST 08, Lillehammer, Norway. 2008.
[MagicDraw]	NoMagic MagicDraw product web page; <a href="http://www.nomagic.com/">http://www.nomagic.com/</a>
[MDTESTER]	MDTester <a href="http://www.fokus.fraunhofer.de/go/utml">http://www.fokus.fraunhofer.de/go/utml</a>
[NSN]	Nokia Siemens Networks web site; <a href="http://www.nokiasiemensnetworks.com">http://www.nokiasiemensnetworks.com</a>
[Neth]	Nethawk EAST product web page; <a href="https://www.nethawk.fi/products/nethawk_simulators">https://www.nethawk.fi/products/nethawk_simulators</a>
[Preevision]	Aquintos PreeVision <a href="http://www.aquintos.info/index.php?menuid=1">http://www.aquintos.info/index.php?menuid=1</a>
[Pro05]	Prowell S. J., Using Markov Chain Usage Models to Test Complex Systems, 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 9 - 2005
[PP03]	S. J. Prowell, J. H. Poore, Foundations of Sequence-Based Specification, IEEE Transactions on Software Engineering, Vol. 29, No. 5, May 2003.
[P09]	Tuomas Pääjärvi. Generating Input for a Test Design Tool from UML Design Models; Master's Thesis; Åbo Akademi University; 2009.
[Ringler]	T. Ringler u.a.: Ein Ansatz für den werkzeuggestützten Elektrik-/Elektronikarchitekturentwurf. ATZelextronik, Januar 2008.
[RSA]	IBM Rational Software Architect. <a href="http://www-01.ibm.com/software/awdtools/architect/swarchitect/">http://www-01.ibm.com/software/awdtools/architect/swarchitect/</a>
[TPT]	PikeTec TPT. <a href="http://www.piketec.com/index.php">http://www.piketec.com/index.php</a>
[TPLan]	ETSI ES 202 553: "Methods for Testing and Specification (MTS); TPLan: A notation for expressing Test Purposes"
[TTWB]	TestingTechnologies. TTworkbench: an Eclipse based TTCN-3 IDE. <a href="http://www.testingtech.de/products/ttwb_intro.php">www.testingtech.de/products/ttwb_intro.php</a>
[TTwb09]	TTWorkbench <a href="http://www.testingtech.com/products/ttworkbench.php">http://www.testingtech.com/products/ttworkbench.php</a>
[SQRL]	SQRL group, University of Tennessee, Link: <a href="http://www.cs.utk.edu/sqrl/">http://www.cs.utk.edu/sqrl/</a>
[SZ06]	Schäuffele, J., Zurawka, T.: Automotive Software Engineering, ISBN: 3528110406. Vieweg, 2006.
[StarUML]	StarUML tool. <a href="http://staruml.sourceforge.net/en/">http://staruml.sourceforge.net/en/</a>
[SysML]	OMG Systems Modeling Language; <a href="http://www.omg.org/spec/SysML/1.1/">http://www.omg.org/spec/SysML/1.1/</a>
[TTM09]	TTModeler. <a href="http://www.testingtech.com/products/ttplugins_modeler.php">http://www.testingtech.com/products/ttplugins_modeler.php</a>
[U2TP04]	U2TP Consortium. UML 2.0 Testing Profile, April 2004. <a href="http://www.omg.org/cgi-bin/doc?ptc/2004-04-02">http://www.omg.org/cgi-bin/doc?ptc/2004-04-02</a> .
[UML]	OMG Unified Modeling Language (UML); <a href="http://www.omg.org/spec/UML/2.1.2/">http://www.omg.org/spec/UML/2.1.2/</a>
[JZN09]	J. Zander-Nowicka.: Model-based Testing of Embedded Systems in the Automotive Domain, PhD Thesis, Technical University Berlin, ISBN: 978-3-8167-7974-2. Fraunhofer IRB Verlag, 2009

	White Paper: V 1.6	Page : 7 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

## EXECUTIVE SUMMARY

This document provides a summary of the common approach on Architectural-Driven Model-based Testing process developed in the D-Mint project. The approach is applied on several case studies within D-MINT, spanning several application domains, ranging from telecommunications to automotive and industrial control. Depending on the specific characteristics of each application domain and case study,, variants (small deviations from the common baseline) can be observed with respect to the techniques, notations, and tools employed at different steps of the MBT process. This document focuses on the commonalities of these variants showing that they use similar techniques, notations, and tools for architecture-driven testing. The main focus of the document is put on the architectural modeling and automated test generation phases.

## 1. INTRODUCTION

Software-intensive systems are playing an increasing role in industrial products such as cars, trains, manufacturing machines, industrial automation systems, mobile phones, and financial computer-supported systems. The complexity of such systems is increasing both in terms of code and algorithms as well as the interconnection of embedded devices. Furthermore, the qualitative importance of the software embedded in these products is growing because of the high amount of safety-relevant controlling functions. Many industrial companies are acting as system integrators, i.e. they assemble system parts (often delivered by different suppliers) to build complete systems. This applies also for the software embedded in these systems, i.e. the companies are also in the role of a software integrator. Since the companies are responsible for the overall quality of the product, they usually pay close attention to the integration and the accompanying integration testing of the systems under test. With this in mind, an effective quality assurance mechanism supporting the software development is indispensable. One of the most important means of quality assurance is testing since it is the only proof under real circumstances.

**Model-based testing** (MBT) comprises promising and systematic approaches to solve these problems. MBT refers to software testing where test cases are derived in whole or in part from a model that describes selected, often structural, functional, sometimes non-functional aspects of a system under test (SUT), e.g. architectural models, behavioural models. A model is usually an abstract, partial representation of the system under test's desired behaviour. The test cases derived from this model are functional tests on the same level of abstraction as the model. Model-based test methods differ in the model being considered, the methods taken for test generation and finally the way the test results are being obtained. The benefit of model-based testing is primarily due to the potential for automation to increase effectiveness and efficiency. For finding appropriate test cases multiple techniques can be applied depending on the model type and test purposes.

The ITEA2-project **D-MINT** aims at developing a new approach to model-based testing which combines various system aspects and models during test generation. This integration of different system aspects and different system models becomes important as system complexity increases. It is also important for systems that are developed partially in software and partially in hardware or that are developed by different vendors with off-the-shelf components. We call this new approach **architecture driven model-based testing**. The architectural model will allow us to combine the aspects, relations and models of the various kinds of system components and thereby provide a unified solution.

The document is structured in the following way. In section 2, the architectural views of the D-Mint approach are described. The generic test process based on the different architectural views and model-based test methods is presented in section 3. Section 4 covers the case-study-specific instantiations of the architecture-driven model-based test process. In the annex, the different MBT methods are described in detail.

## 2. ARCHITECTURE-DRIVEN MODEL-BASED TESTING PROCESS IN D-MINT

Looking at the system from different points of view is a way to deal with complexity. We could for example look at our system from different viewpoints. A *viewpoint* depicts the system in the form of blocks that realize the required functionality without taking into account any technical aspects of their realization.

Within our approach we distinguish the following architectural views (as depicted in Figure 1):

- Functional View
- Logical Architectural View
- Technical Architectural View

- Topology Architectural View

Within the different case studies, all views or only parts of them may be used. In order to get an impression of what information is collected under each of these views and how they are interrelated, we will provide a short presentation of each view in the system design phase.

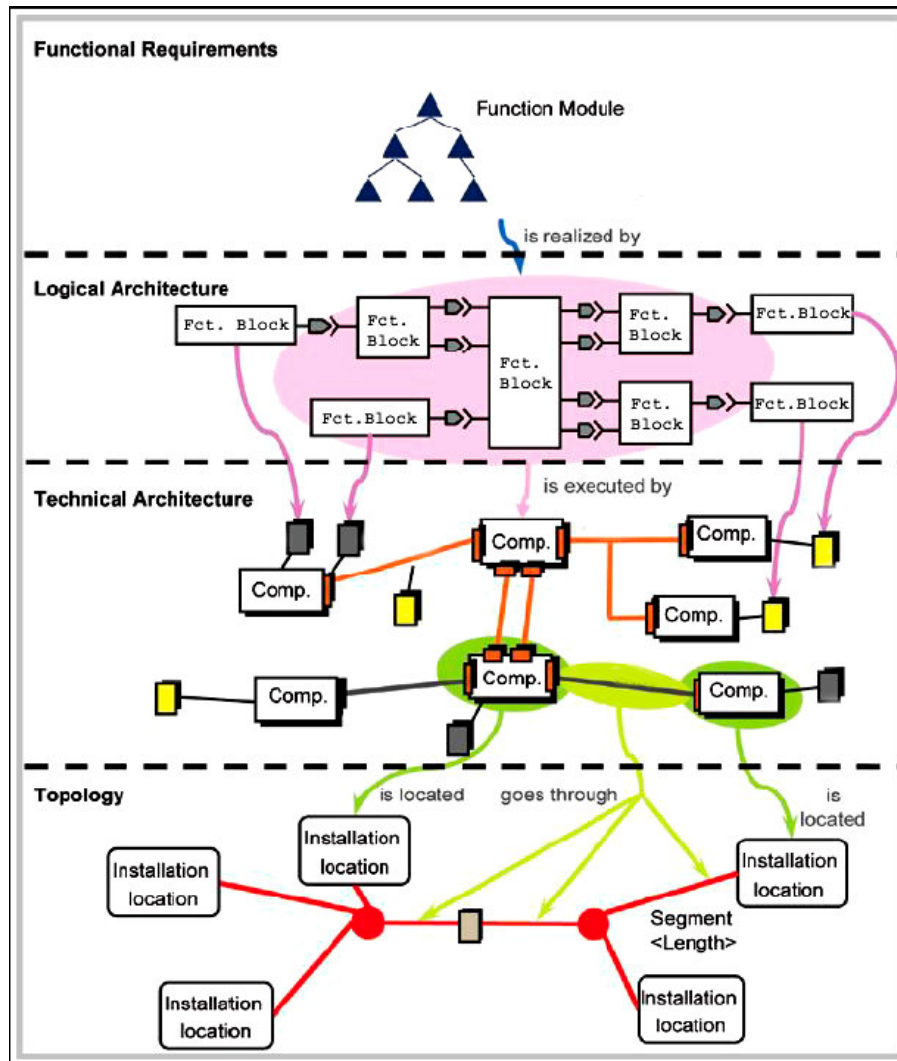



Figure 1 Architecture-driven MBT viewpoints [Ringler]

## 2.1 FUNCTIONAL VIEW

Within the **Functional View** the *functional requirements* are presented. These requirements are described from the perspective of their “usage” interaction with the system and they define how the system is expected to behave when certain actions are stimulated by its “users”. The functional requirements do not include requirements specific to components or parts of the system. The functional view regards the whole system as a black box and only the interaction with the user is visualized. The functional requirements are structured in a tree-like structure, giving a hierarchical, functional grouping.

	White Paper: V 1.6	Page : 10 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

## 2.2 LOGICAL VIEW

The **Logical Architectural View** is used to describe the system as a composition of interacting, *functional blocks* without relating to their technical realization, e.g. completely in hardware or as software running on specific hardware. The modeling elements in this view are *functional blocks*, *interfaces / ports* through which the functional blocks can communicate with each other using operational *calls*, *messages*, *events*, or *continuous signals*. Specific blocks can be used for the “interaction” with the environment: e.g. sensors and actuators. The modeling elements, including the interchanged information, are defined as types in this view and also as interconnected instances of these types, representing the system and realizing the functionality defined in the functional view.

Within the logical view a decomposition of the system into functional blocks takes place. The functions required in the functional view from the perspective of the system use are to be provided in the logical view by a cooperation of functional blocks.

The logical view can be extended with additional requirements for each component of the system. These extra requirements are refinements from the functional requirements and describe how the individual parts of the system are supposed to contribute to the main functionality described in the functional view.

The requirements specified in the logical view need to be associated to the functional requirements. These associations have to be specified in the model as well. While the functional requirements can be used to generate integration tests or interoperability tests, the requirements given in the logical view can be used to generate conformance or unit tests.


## 2.3 TECHNICAL VIEW

The **Technical View** contains details on technical aspects related to hardware and software characteristics. It describes the realization of logical view functional blocks using a specific hardware. For instance, for an embedded system composed of ECUs, sensors, actuators and busses, the technical view should contain details on the assignment of functional blocks to ECUs, interconnection of ECUs by busses, connections of sensors to ECUs etc. For a system in the telecommunication domain which may consist of several interconnected servers, the technical view should contain details on the servers interconnected, software configuration, hardware configuration, connections, settings (e.g., IP addresses, ports), parameters (topology hiding enabled), etc. As a rule of thumb, the technical view should definitely contain all possible configurations which appear as preconditions of requirements in the functional view. In case different configurations conflict with each other, the architectural view should contain different diagrams such the engineer can refer to one or the other configuration by simply indicating the diagram.

Similar to the decomposition of the requirements from the functional view, in the logical view we get a set of technical view requirements, as a set of requirements to the technical view components, derived from the original functional view requirements. The technical view may also contain further requirements which can be used for test generation. These requirements also need to be related (as refinements) to the requirements from the other two views.

## 2.4 TOPOLOGY VIEW

The **Topology View** describes the *topological* and *geometrical* aspects of the system. The topology information should not be confused with network topology which is already described in the technical view. In this view, the topology information only concerns the *spatial placement* of hardware devices (e.g., placement of ECUs in a car and the wiring). For many systems this kind of information plays an important role in the functionality of the system. For example, the blinking actuators need to be placed in the right

	White Paper: V 1.6	Page : 11 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

locations. The front left blinker should be placed exactly in the front left side of the car. It makes sense to test whether the location is correct as it may happen that the system is working correctly but only the wrong location makes the system not to be functional. Such tests obviously require special equipments in order to be able to perform topological observations.

Another example comes from the telecommunication area. A system with a mesh of wireless nodes may associate client terminals to receiving antennas simply based on the distance between terminals and nodes. This information is therefore relevant to testing too, as the test system should provide this spatial topological configuration before running the actual test. However, many systems do not have or do not require such information or the topology placement is not even important at all.

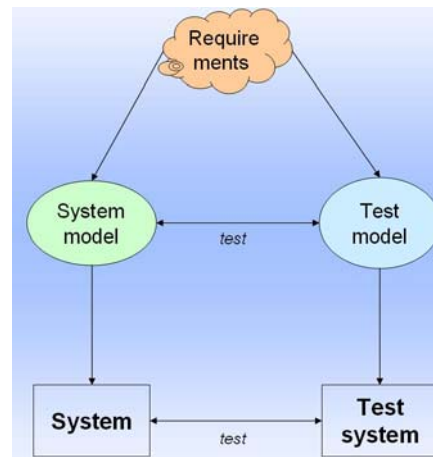
### **3. ARCHITECTURE-DRIVEN APPROACH AS A TEST CREATION METHODOLOGY**

The D-Mint Common Approach applied within the different case studies describes the test process from the requirements to the reporting of the test results in six different steps (Figure 2). There are some deviations from this process in some case studies since in those case studies not all steps are applied, e.g. the ETSI case study does only specify tests but does not execute them. They are only interested in the specification of tests for a specific standard, which may be executed separately outside ETSI. In addition, due to the rather heterogeneous set of domains targeted in this project (from standardization to automotive to telecommunication), the set of notations and tools that will vary. Nevertheless, the methods are quite similar and even sometimes the set of tools and notations.

<b>PROCESS</b>	<b>5.1</b> Requirements Management & Documentation	<b>5.2</b> Modeling for Test Derivation	<b>5.3</b> Test Derivation	<b>5.4</b> Test Implementation	<b>5.5</b> Test Execution	<b>5.6</b> Test Reporting
<b>ABSTRAC-TION</b>	<b>2</b> Abstraction Level: System Architecture Viewpoints: Requirements (all), Logical (all), Technical (most), Topological (possibly of interest, but not realized)					
<b>METHODS</b>	<b>5.1.1</b> Structured Requirements (Sometimes up to "Precondition, Event, Reaction" formal level)	<b>5.2.1.1/5.2.2.1</b> Architecture based; Behavior modeled by state chart, sequence charts or signal flows; Priority by annotations of usage, risk, safety, quality validation	<b>5.3.1.1/5.3.2.1</b> Test cases derived from architecture models and behavior with respect to annotated priorities and coverage criteria	<b>5.4.1</b> Abstract test cases in a test model or test specific language "compiled" to some byte code – some times only test descriptions for manual execution	<b>5.5.1</b> Online, offline, HIL, SIL, PIL	<b>5.6.1</b> Pass/Fail; Statistical analysis; Test execution traces; Back-tracing of Req's
<b>NOTATION</b>	<b>5.1.2</b> Textual format, tabular format, sequence-based specification, Standardized test specifications (ETSI), SysML	<b>5.2.1.2/5.2.2.2</b> UML, SYSML, OCL, MSC, TPT, PTML Domain specific languages, Model Annotations for priority	<b>5.3.1.2/5.3.2.2</b> QML TTCN-3 Tool specific	<b>5.4.2</b> QML/EAST TTCN3	<b>5.5.2</b> Machine code java byte code EAST scripts	<b>5.6.2</b> HTML+UML
<b>TOOLS</b>	<b>5.1.3</b> Text based tools, DOORS, PreeVision, MagicDraw (UML)	<b>5.2.1.3/5.2.2.3</b> MagicDraw, EA, StarUML TTModeler, MySQL, Juml, TPT, Qronic Modeler, PreeVision, MATERA, Preevision, RSA	<b>5.3.1.3/5.3.2.3</b> Qronic, TTModeler, MOTES	<b>5.4.3</b> Qtronic TTWorkbench Juml TPT TTCN-3 Express (PreeVision, OpenOffice)	<b>5.5.3</b> Qtronic EAST TTWorkbench iXtronics Testrig TTCN-3 Express, MessageMagic	<b>5.6.3</b> Qtronic EAST TTworkbench TTCN-3 Express

**Figure 2 The D-Mint Common Approach**

All case studies start with the *Requirements Documentation* step in some textual form, sometimes in a quite formal way of description. From these requirements, a model or a set of models of the SUT are created during the Modeling for Test Derivation step. From the resulting models (which can be either *test models* or *system models*) tests are derived in the *Test Derivation* step. This may be done a) through a System Model, b) through a Test Model, or c) on the way from the Requirement model through the System Model and a Test Model (see Figure 3). Abstract test cases are derived from the models and are then "transformed" into executable test cases during the Test Implementation step. Test Execution may be performed manually or automatically. Sometimes the tests are run against the real "machine" sometimes against a simulation model of it. The Test Reporting step may comprise quite simple pass/fail reporting of the tests, ranging from statistical analysis of the resulting test traces to in depth backtracking of the results to the requirements. Reporting is mostly done within the case studies in a textual form using HTML.



**Figure 3 System Model vs. Test Model test generation<sup>1</sup>**

The level of abstraction applied within the case studies is that of the System Architecture. In most case studies, “Viewpoints” have been used to focus on specific aspects of the system and to deal with the complexity of the system. Viewpoints also support the reuse and testing of parts of the system (e.g. software) independent of its use in combination with specific hardware (e.g. the processor, server, ECU) it is afterwards used on.

The specific use of the process in the context of the case studies is described in Section 4. More concrete details regarding various methods, notations, and tools described in Figure 2 are given in the Annex. In the following, we give a brief overview of the commonalities within the D-Mint Common Approach.


### **3.1 REQUIREMENTS MANAGEMENT AND DOCUMENTATION**

Requirement management and documentation are crucial aspects in contemporary product development as they allow the specification of what the customer expects from the product, what features the product has, and how these features should be realized in practice. The requirements will accompany the product development through all its phases and traditionally they become input for the testing process. Empirically it has been observed that around half of the errors detected during the testing phase are due to a poor requirement management and documentation process.

The approaches followed in the D-MINT project with respect to requirements management and documentation are discussed in Annex 5.1. All the evaluated case studies within D-MINT use some kind of method and the supporting tools for the requirements management and documentation, spanning textual notations, structured documents, tabular notations, and ending with graphical modeling language like the System Modeling Language (SysML) [SysML]. Moreover, requirements are propagated in all case studies though the various phases of the MBT process and traced to different artifacts like system/test models and test case specifications, respectively.

### **3.2 MODELING FOR TEST DERIVATION**

<sup>1</sup> The difference between system and test models with respect to MBT is that the former is a model describing the behavior of the SUT from an internal perspective, whereas the later describes the behavior of the SUT from the perspective of the environment.

	White Paper: V 1.6	Page : 14 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

Models play a pivotal role in model-based testing as they are used as the source for automatic test derivation. Therefore, the type and quality of the information included in these models greatly influence the quality of the generated tests. Models have become increasingly popular in software development and they started to become main stream artifacts in many industrial environments, as well. Their popularity is largely due to the abstraction level and graphical notations they provide, as well as to the increasing capabilities of the modelling tools towards model management, validation, and code generation. In addition, there are nowadays available a multitude of models types that can be used for modeling a (software) system depending on the perspective the user wants to focus on, for instance state charts, Markov chains, class diagrams, deployment diagrams, Petri Nets etc. All these aspects enable for the usage of models to bring important benefits over the traditional testing process.


As pointed out in the beginning of Section 3, there are two distinct approaches followed in D-MINT, differentiated by the nature of the model used for test derivation. A first approach, considers a *system model* of the SUT which is similar with the specification of the system (on a certain abstraction level) used in the development process. In such case, the behavior of the SUT is described from an internal perspective, i.e. what input or output the SUT can receive or respectively, provide to the environment in each state. The methods and tools used in this approach within the D-MINT case studies resemble those of the current software development practices: the SUT is specified from different perspectives (functional, behavioral, *architectural*, data-oriented) by employing graphical modeling languages provided by the model-driven architecture (MDA) community, like the Unified Modeling Language (UML) [UML]. Similarly, already existing (UML-based) modeling tools or extensions to them are used to support the approaches in order to fully benefit from capabilities like graphical editors, model management, versioning, model validation, and model transformation or generation engines. In certain cases within, existing modeling languages are customized to include annotations for priority and probability. A more detailed description of the methods, languages and tools used in D-MINT is given in Annex 5.2.1.

In contrast, the second approach considers a *test model* for test generation, in which the behavior of the SUT is specified from an external perspective, that is what output the SUT provides to a given input under certain conditions of the environment. Traditionally, such models are also referred to as usage models. This approach is more similar to the traditional way of testing. Again, the SUT is specified from several perspectives, in which the *architecture* plays a central role. As the nature of the models employed in this approach is different than that of the traditional modeling languages, customizations of the later are performed. Generally, existing formalisms like message sequence charts, Markov-chains, sequence based specifications or customizations of UML (via profiles or metamodeling techniques) have been used within D-MINT for addressing aspects specific to different application domains, by introducing concepts like time, probabilities, test architecture, etc. A more detailed description of the methods, languages and tools used in D-MINT is given in Annex 5.2.2.

### 3.3 TEST DERIVATION

Test derivation is a key issue within the automated test development process. The power and benefit of test derivation algorithms depend on the level of detail and the completeness of the models to be used. Test derivation addresses the production of test cases that cover the original model, e.g. usage scenarios, requirements or architecture elements. The derivation algorithms apply various test techniques based on coverage criteria (e.g. data partitioning, state-based coverage) or viewpoint combination (e.g. global/local or logical/technical views).

Methods, notations and tools used in this context in D-MINT have been introduced in Annex 5.3. Although the test derivation can receive as input either system models or test models, the principles applied for test derivation are similar. Pure, annotated or profiled UML is mostly used as the model notation. Although test description formats for resulting test cases (i.e. the tool output) can be different there is some common trend that the tool output is provided using a TTCN-3 notation. It is due to the abstraction level that makes TTCN-3 a kind of test middleware.

	<p>White Paper: V 1.6</p>	<p>Page : 15 of 51</p>
		<p>Version: 1.6 Date : 06/11/2009</p>
		<p>Status : Final/Released Confid : Public</p>

Since test derivation methods often fail due to an unrealistic number of generated test cases the application of priority calculations is popular in order to get a means for selecting or ordering of test cases for test execution. They address different criteria like usage probabilities, costs, or risks. Some techniques like statistical-based test generation already include such indicators and calculations, other model-based approaches like architecture-driven testing are going to be extended to include and consider such enhancements.

### 3.4 TEST IMPLEMENTATION

Test implementation addresses the test development step from abstract to executable test cases and the provision of a complete test system ready for test execution. The generated abstract test cases (specified using either graphical or textual notations) need to be enriched (e.g. with configuration data) and adapted to the concrete SUT interfaces in order to be executed within real test campaign. Since this is not a specific problem in the MBT area here it is only briefly addressed. Existing approaches are using compilers/translators towards executable notations and the addition of SUT specific utilities like data codecs and interface adapters. Details of some technical solution applied in D-MINT are given in Annex 5.4. Some tools provide different translation methods (online/offline) and several options for backend scripts. Additionally, various approaches have been applied in order to increase the automation of this step, e.g. using predefined reference libraries, utilities to generate test adapters/codecs, or virtual machines.

### 3.5 TEST EXECUTION

The execution of the test cases generated using MBT techniques requires regular means of testing for the real test campaign. Test execution is pursued either following the traditional way, e.g. the test cases are run post-generation against the SUT (offline testing), or by executing them while being generated (online testing). In addition, the availability of the SUT or of the test models is used in certain cases for running the tests against the system simulation or performing additional activities during test execution, e.g. online result evaluation against some underlying model. Tool support for test execution provides a long list of common test execution features like optimization of the test execution performance, test management, user assistance during test execution (e.g. debugging, backtracking, and quick fixes), stress/load testing and graphical integrated development environment capabilities. Further details on such capabilities are addressed in Annex 5.5.

### 3.6 TEST REPORTING

Test reporting addresses the presentation and analysis of test results. The approaches followed in D-MINT vary from a simple list of verdicts, the production of test log traces in various formats, to backtracing to test case definitions, models or even requirements. It may also address user-friendly highlighting of system elements and/or (requirement) coverage statistics. In certain cases, the test reporting is used as source for root-cause analysis (i.e., detecting the fundamental reason why a test failed). Further details are provided in Annex 5.6.

## 4. THE COMMON APPROACH BASED ON THE SELECTED CASE STUDIES

In this section selected phases of the creation of the test specification for a number of D-MINT case studies are described. These phases comprise:

- Requirements Specification
- Model for Test Derivation
- Test Derivation/Generation
- Test Implementation

- Test Execution
- Test Reporting

All of them are defined having the perspectives of architectural abstraction, methods, notation, and tools in mind.

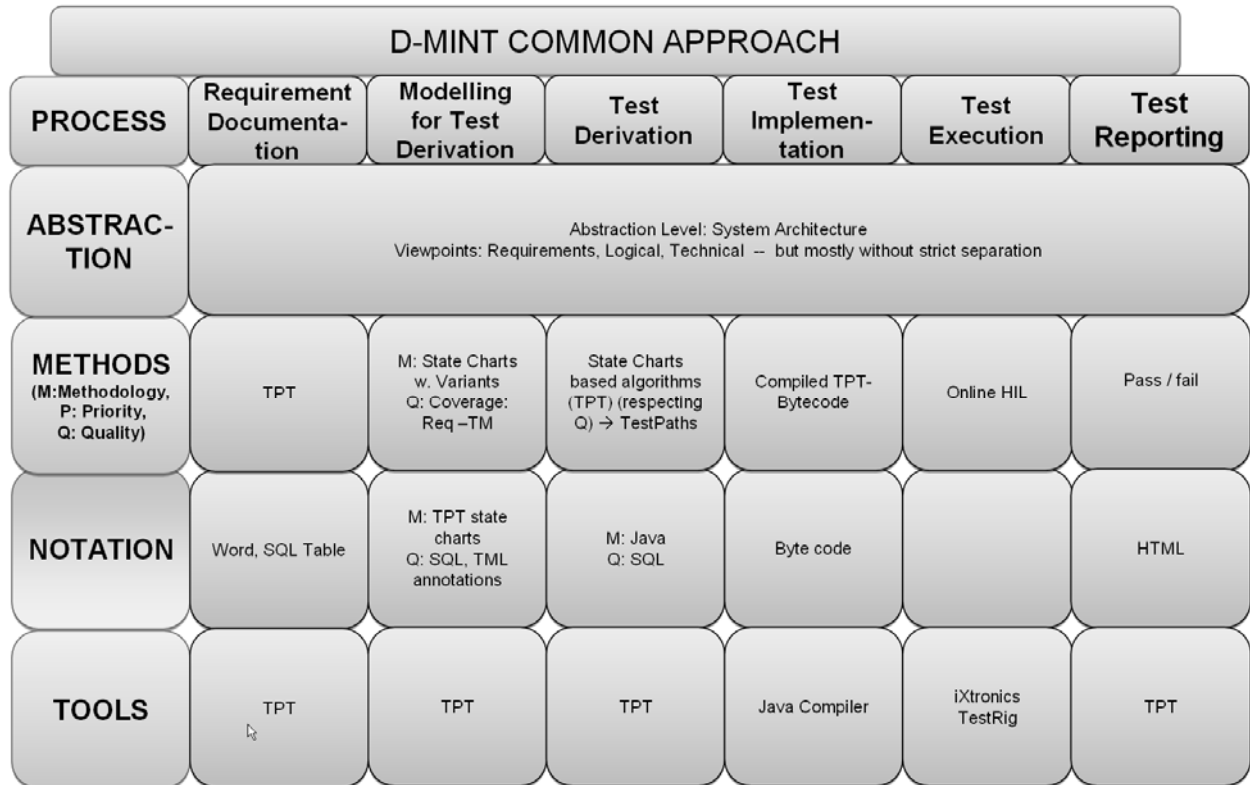
#### 4.1 ABB CASE STUDY

ABB case study is analyzed and tested in three different ways. First, it is performed using IESE methodology. Then, TPT/CamelView approach is applied. Finally, TTmodeller approach validates the system.

D-MINT COMMON APPROACH						
PROCESS	Requirement Documentation	Modelling for Test Derivation	Test Derivation	Test Implementation	Test Execution	Test Reporting
ABSTRACTION	Abstraction Level: System Architecture Viewpoints: Requirements, Logical, Technical -- but mostly without strict separation					
METHODS (M: Methodology, P: Priority, Q: Quality)	SBS->FSM (finite state machine)	M: Markov chains (annotated FMS) P: Usage, Risk, Safety annotat. Q: Coverage: Req – TM	Structural (TM coverage->O) + Statistical (random according to P) -> TM paths	Each test case is mapped to a script test scripts (textual format: as C-code or ttcn3,...)-> Compiled scripts	Offline HIL	Pass/fail after stimulus, failing state, statistics, reliability estimations
NOTATION	Word, SQL Table	M: Usage profile for TML P: Annotations in TML Q: TML annotations	M: Markov chain based alg. -> TM paths (from START to EXIT state) P: Interpretation of TML	Test scripts -> Byte / machine code		HTML
TOOLS	Protoseq, jSeq	yED, Texteditor, EA	jumb1	Execution on target platform (no specific test execution env.)	iXtronics TestRig	jumb1

Figure 4 The ABB / IESE Case Study

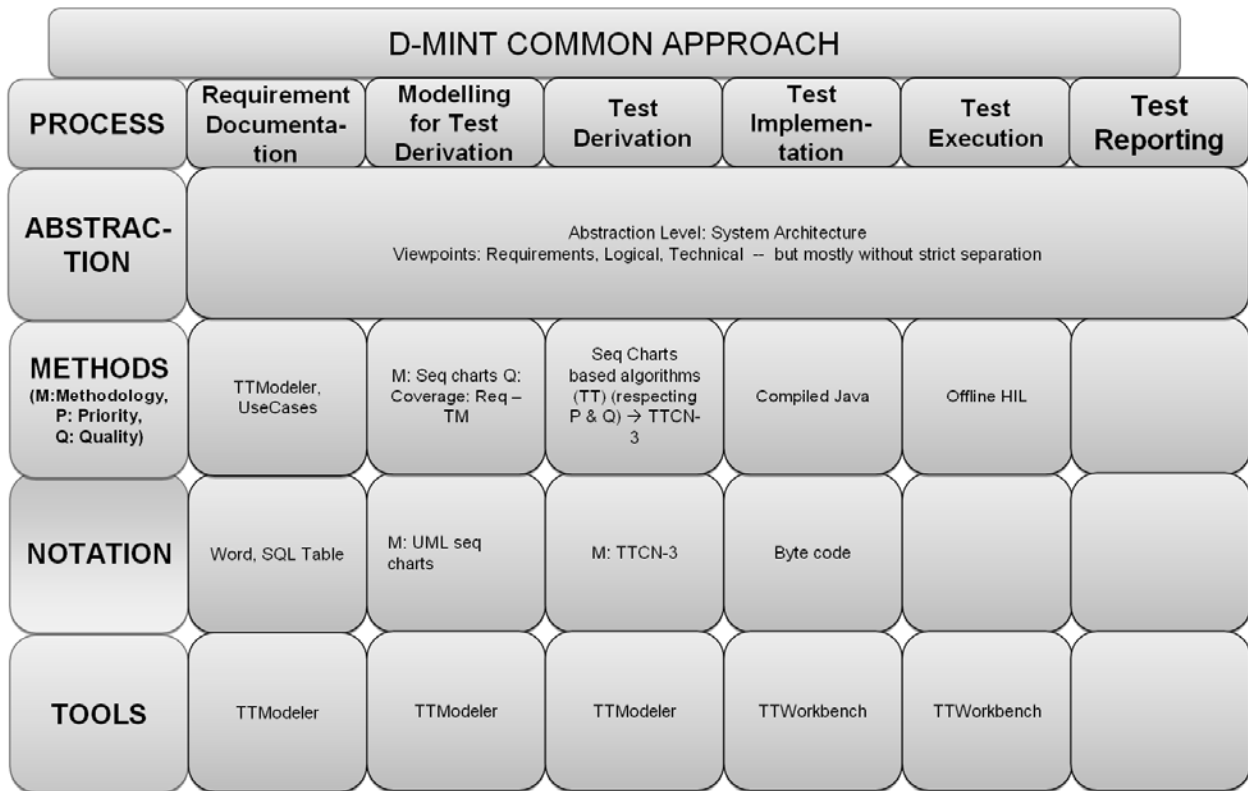
Using the **IESE methodology** [BAU09], black-box test cases for the system under test (i.e., soft starter) are generated using statistical state charts that represent the test model. ABB case study purpose is to enhance the manual manner of defining test cases with the development of a test model that is based on documents such as requirements or user manuals. These requirements are manually transformed into sequences describing the behavior of the system (i.e., they are called Sequence Based Specification, SBS). These sequences are then automatically translated into state charts using ProtoSeq. In addition to the bare state chart, typical usage scenarios, a risk distribution and the safety assessment for the system components are needed to develop profiles for the probabilities of the transitions in the state chart. With these probabilities and the state chart the tool JUMBL is able to derive test case sets for each profile. Currently the test cases are generated as C-Code conforming to the test interface developed in the lines of this project. Then, they are executed on iXtronics TestRig, either as textual notation for manual execution, or as TTCN-3 code. The analysis of the test results is done with JUMBL by feeding back the textual result files.



**Figure 5 The ABB / TPT case Study**

Using **Time Partitioned Testing** (TPT) [TPT] test cases are defined by state charts with a variable behavior. Each state can refer to a different behavior (e.g., different C-functions of the test interface). Thus, a single state chart describes a set of test cases that is given by the permutation of all possible behaviors in the states. Which test cases to choose is still a development task, although structuring/clustering the test cases, modeling the test cases with state charts, and reading as well as processing result values is well supported by TPT.

TPT is integrated with iXtronics CAMElView/TestRig to enable the execution of test cases and reading the back measured values for the analysis of test case results.

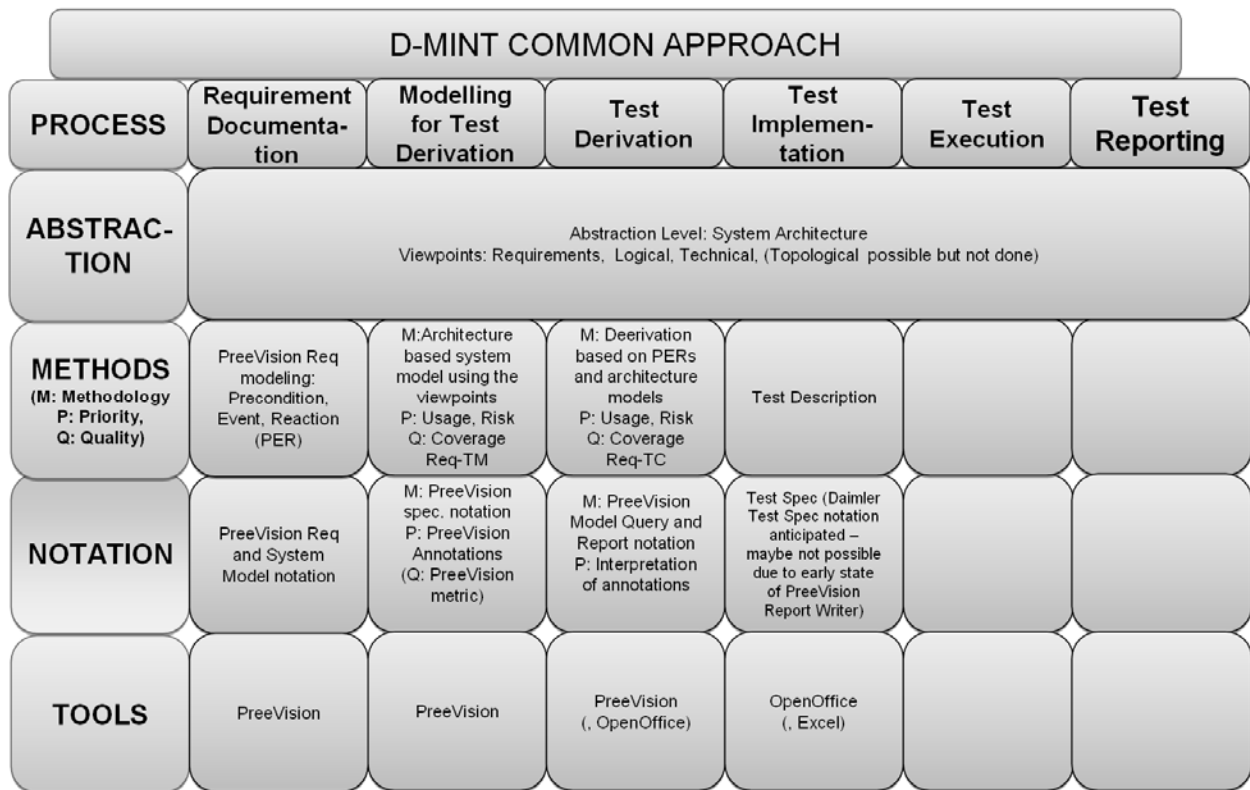


**Figure 6 The ABB / TTModeler Case Study**

Finally, **TTModeler** supports the application of the UML Testing Profile. It enables modeling of test cases by sequence diagrams. Again, which test cases to select is still a development task. Structuring and further processing of the test cases is supported by the tool. Once the test cases are developed they are transformed to TTCN-3, which can then be executed in a test environment with the corresponding SUT.

## 4.2 DAIMLER CASE STUDY

In the DAIMLER case study two different approaches are applied on two distinct case studies as discussed in the following.



**Figure 7 The Daimler / FOKUS Case Study**

**The Daimler/FOKUS case study** starts with an experimental architectural model of the exterior mirror of a car, build with PreeVision by Daimler. The goal of the case study is to use the information kept in the system architecture model to automatically derive test descriptions and even to derive new kinds of tests and aspects not captured before.

The case study uses *three views on the system architecture*. *Requirements* describe the required system functionality in the form of Precondition, Event, Reaction triples in a very strict way. In the *logical* view the system functional blocks are described without any information on actual physical or technical realization aspects. In the *technical* view the system architecture is described by its technical realization blocks. Technical aspects such as, for example, the collocation of different logical functions on the same hardware or the combined use of the same communication channel are modeled at this level. In the *topological view*, which will not be investigated in this case study the locations for hardware installation, or wiring etc. will be modeled in the system architecture.

It is important that links between different system elements exist in every of the views. The requirement parts (i.e. functions, preconditions, events, reaction) are linked to the functional blocks, ports, and communication channels that realize them in the logical view. The logical functional blocks are linked to the technical blocks that realize them. So, we can use these relationships for testing the required functionality in many environmental realization contexts.

The *requirements specification* is provided in the form of precondition, event, reaction triples for each function. Functions can be grouped. Functions, preconditions, events, and reactions are linked to the architecture model elements in the logical architecture view. The applied notation is specific to the PreeVision tool.

The system architecture model with its different views including the textual requirements description serves as a source *model for test derivation*. This is done based on a set of algorithms described in the case study. A separate test model would have been useful here as well. Though, due to the restrictions of PreeVision we derive the tests straightforward from the system model.

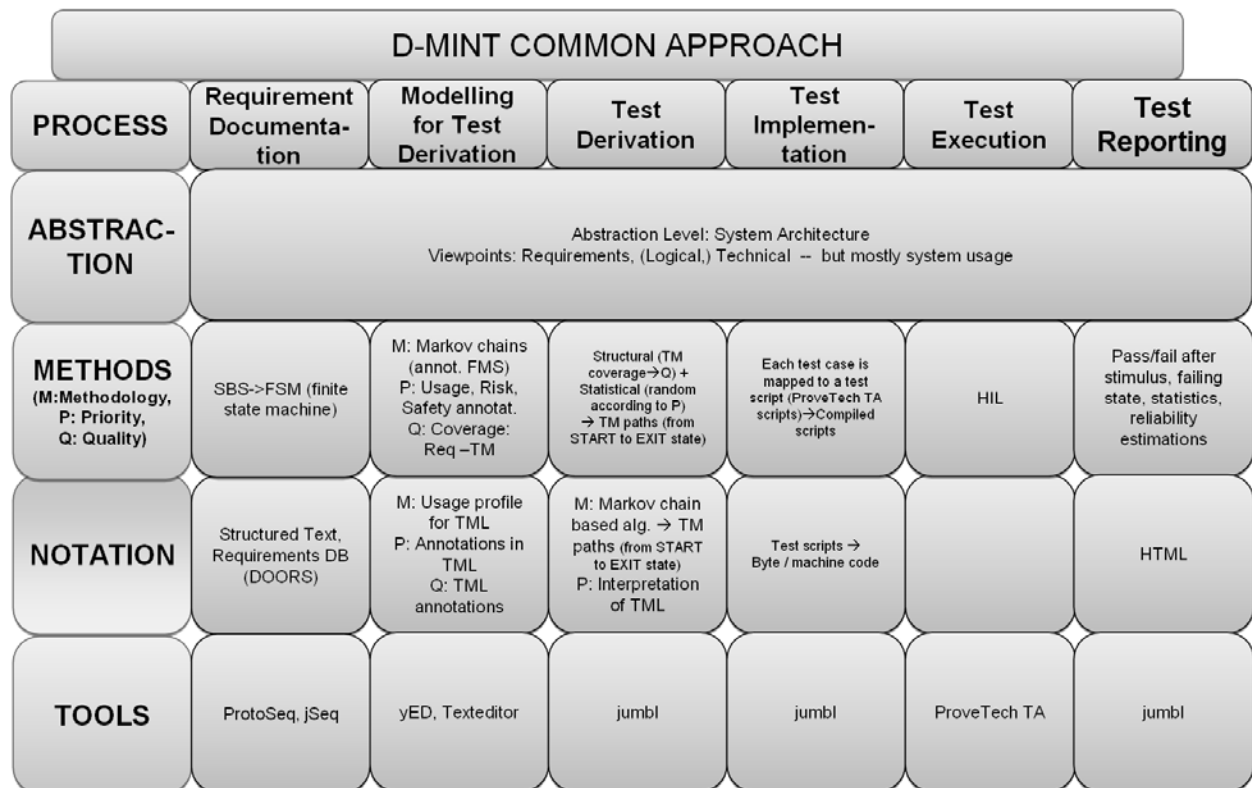
Priorities concerning the usage or risk of functions can be annotated with the model elements in the system architecture model in all the views. The notation is specific to the PreeVision tool.

A set of algorithms to *derive tests* based on the system architecture has been defined in the case study. Examples of these algorithms have been implemented using the model query facilities and report writing facilities available in PreeVision to derive the test descriptions. An intermediate test model and the implementation of the algorithms using this intermediate step would have given more flexibility in the arrangement of those test description reports. Using the tree structure for the report writing and being bound to implement the control logic for the queries in the report writer limited the flexibility.

Priority annotations can be considered for the selection of test cases in the test description generation phase. Again an internal notation of PreeVision has been used to implement the reports. Here, the built in model query facility and Open Office as a report writer integrated in PreeVision were applied.

In the concrete *implementation* of the test derivation algorithms the tests are generated as textual test descriptions. There is no plan to transform them into an automatically executable format in the context of the case study.

*Test execution* and *test reporting* as such are not considered in this approach.



**Figure 8 The Daimler / IESE Case Study**

The test object of the **Daimler/IESE** case study is the blinker control unit (BCU). The BCU is responsible for all blinking functions in the car (e.g. turn light blinking, warn blinking). The test object consists of

several electronic control units which are connected via communication buses. Daimler provided the original requirements specification of the BCU which contained textual, tabular, and graphical descriptions of the BCU requirements.

The goal is the automated generation of test cases by using a separate test model. Here, we apply model-based statistical testing which incorporates usage and criticality profiles to drive the test case generation. The test model is systematically constructed from the original system requirements by applying the sequence-based specification technique (SBS) and the jSeq tool. In the SBS the system boundary, its inputs, input sequences, and responses have been identified and analyzed. The result is black box specification describing all relevant input-output sequences of the BCU. It can be seen as a formalized representation of the functional system requirements.

Test cases are generated from the test model by applying model coverage and random test case generation. The model coverage algorithm ensures the coverage of all states and transitions of the test model. Random test cases are derived according to the usage profile of the test object. This leads to more realistic test cases and more meaningful test results. For the test generation step the tool JUMBL was used. The output format of the test cases complies with a Daimler-specific template for test case descriptions.

The execution and evaluation of test cases are not considered.

### 4.3 ELIKO CASE STUDY

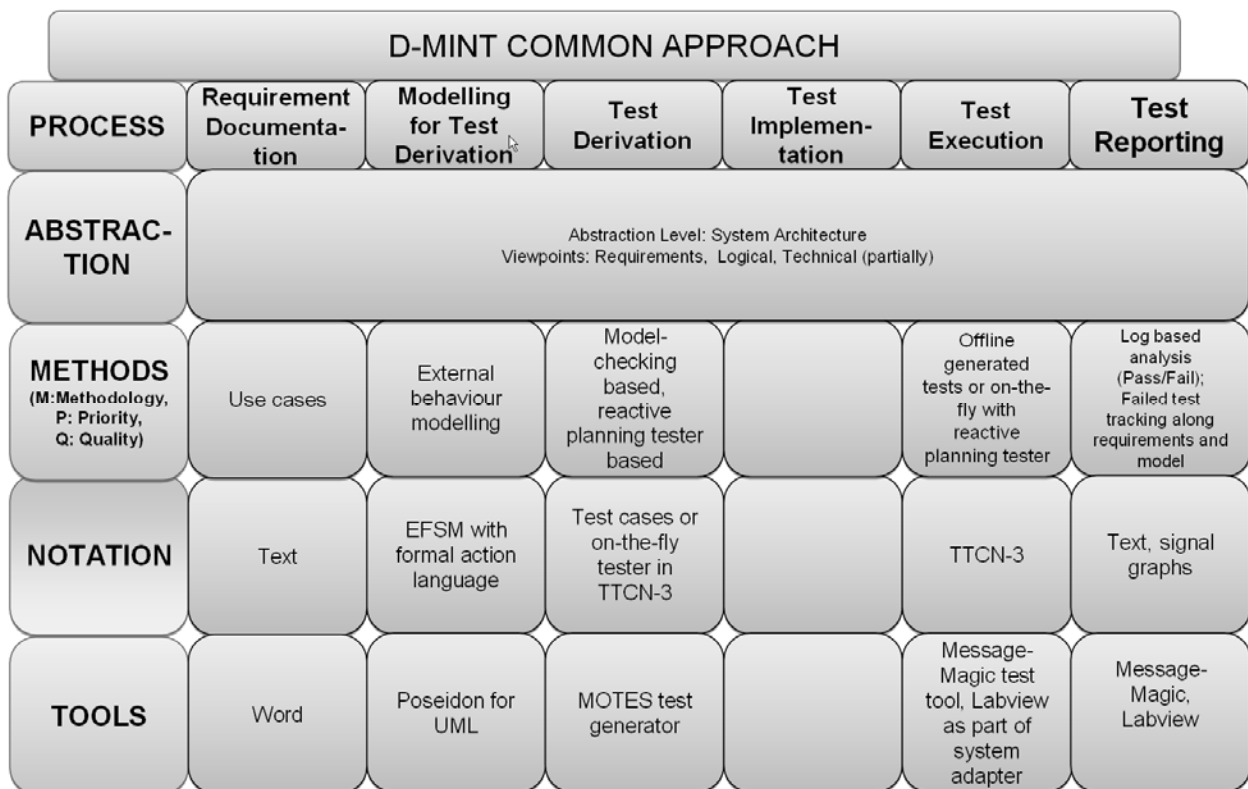



Figure 9 The Eliko Case Study

ELIKO case study applies to the telematics domain. The implementation under test (IUT) forming the case study is a Feeder Box Controller of a street lighting system.

	White Paper: V 1.6	Page : 22 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

The *requirements* of the feeder box controller are specified by a requirements specification document using natural language use cases in English, enriched with diagrams and tables.

*Models for test derivation* are defined in UML state machines. The models define the system under test with respect to expected behavior. The UML state machines are drawn using Poseidon for UML CASE tool. The UML state machines are exported from Poseidon and imported to Elvior MOTES test generator in XMI format before the tests are generated. Context variables, test configuration, and test data are defined in TTCN-3 files. The TTCN-3 files are also imported to MOTES before tests generation.

*Tests are implemented* automatically using MOTES test generator. MOTES takes UML state machines and TTCN-3 files as input and produces TTCN-3 tests. The *test generation* phase can be controlled using various model structural coverage criteria like all transitions, selected states/transitions. From deterministic model of the IUT the test cases in TTCN-3 are generated that implement predefined test sequences. From nondeterministic model of the IUT the reactive planning tester in TTCN-3 is generated. Reactive planning tester generates tests on-the-fly depending on the defined test coverage and on the nondeterministic behavior of the IUT.

The generated test cases (from deterministic models) and the reactive planning tester (from nondeterministic models) are *executed* against the IUT using Elvior MessageMagic TTCN-3 test tool. MessageMagic produces *test reports* in text and XML format. The reports contain, in addition to regular verdict information (pass/fail), the information about the covered states and transitions in the model. The information can be used to trace back to the UML state machines to speed up finding the potential sources of errors.

#### 4.4 ETSI CASE STUDY

The ETSI case study targets the telecommunications domain, with systems implementing the IP Multimedia Subsystem (IMS) standard or combinations thereof building the SUT.

The bases for *requirements specification* in the ETSI case study are the IMS standard documents. Testability requirements are extracted from those standard documents in the form of a document called *test suite structure and test purposes* (TSS&TPs). The test suite structure describes how the test suite is organized (e.g., groups of test cases, targeted SUT features), whereas the test purposes (TPs) describe what functionality each of the test cases will assess on the SUT. Those test purposes are expressed in a notation called Test Purpose Language (TPLan). TPLan is a (semi-formal) standardized notation designed by ETSI for expressing test purposes. Additionally, the TPs in this ETSI case study describe a sequence of actions and events to be performed or observed in order to assess each TP.

The *model for test derivation* is a state automaton of the SUT's behavior, based on the IMS standard documents and on the sequences of actions and events described in the TSS&TP document. That model is expressed in the QML notation designed by Conformiq for the Qtronic tool. Based on that model, the Qtronic tool automatically generates test sequence diagrams, which can then be either automatically or manually transformed into executable test scripts.

To facilitate the *transformation* of test sequences into executable test scripts FOKUS has developed the MDTester tool [MDTESTER] which supports pattern-oriented test engineering using a model-driven approach. The test sequences are modeled in a notation called Unified Test Modeling Language (UTML), which refines and extends concepts of the UML Testing Profile (UTP). The UTML model provides a *requirement view*, a *topological view* and a *logical view* on a combination of the test system and the elements of the SUT it interacts with. The *requirements view* is realized through a test objectives model, while the *topological* and the *logical views* are realized through a test architecture and a test behavior

model respectively. The UTML model allows elements of those different views to be linked to each other into a dedicated test model.

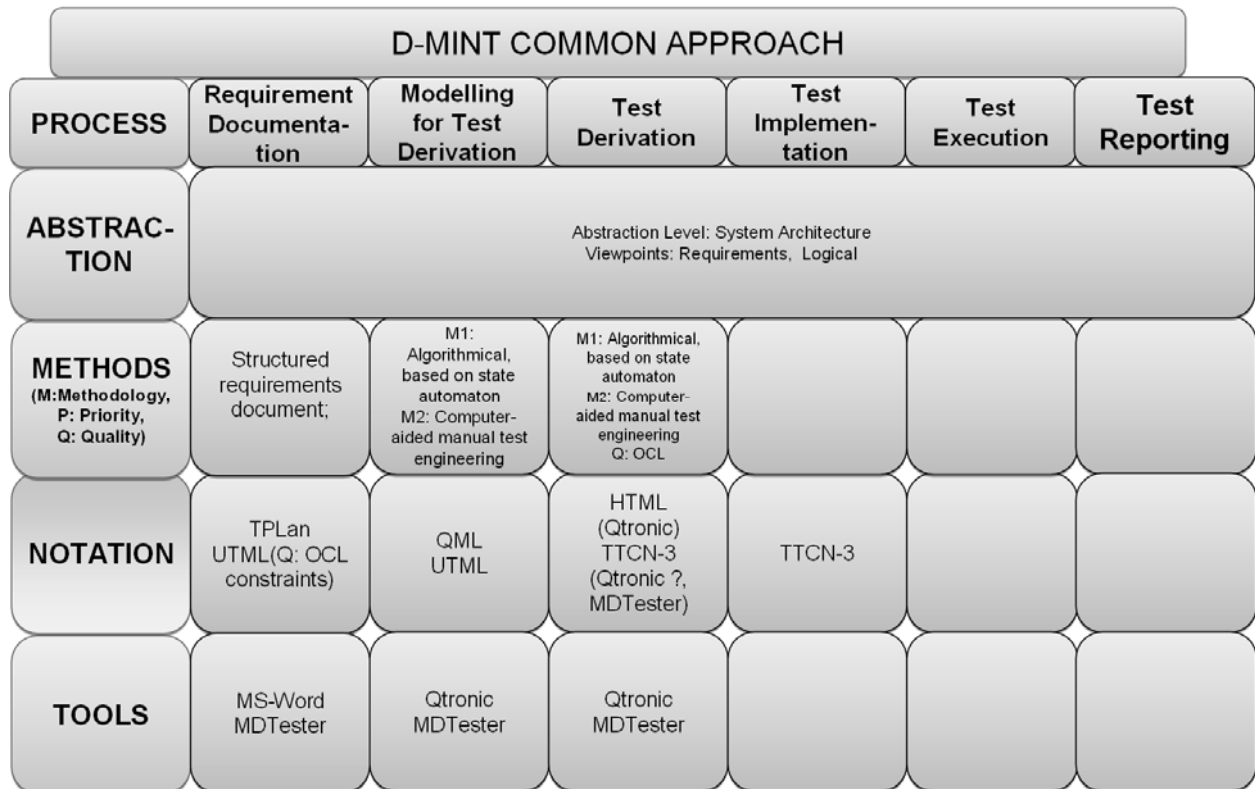


Figure 10 The ETSI Case Study

The UTML test model is transformed automatically into TTCN-3 source code, using a template-based model-to-text transformation. In a further step, the *implementation of the test cases* can be refined, taking the automatically generated test cases as a starting point.

*Test execution and reporting* were not considered for this case study.

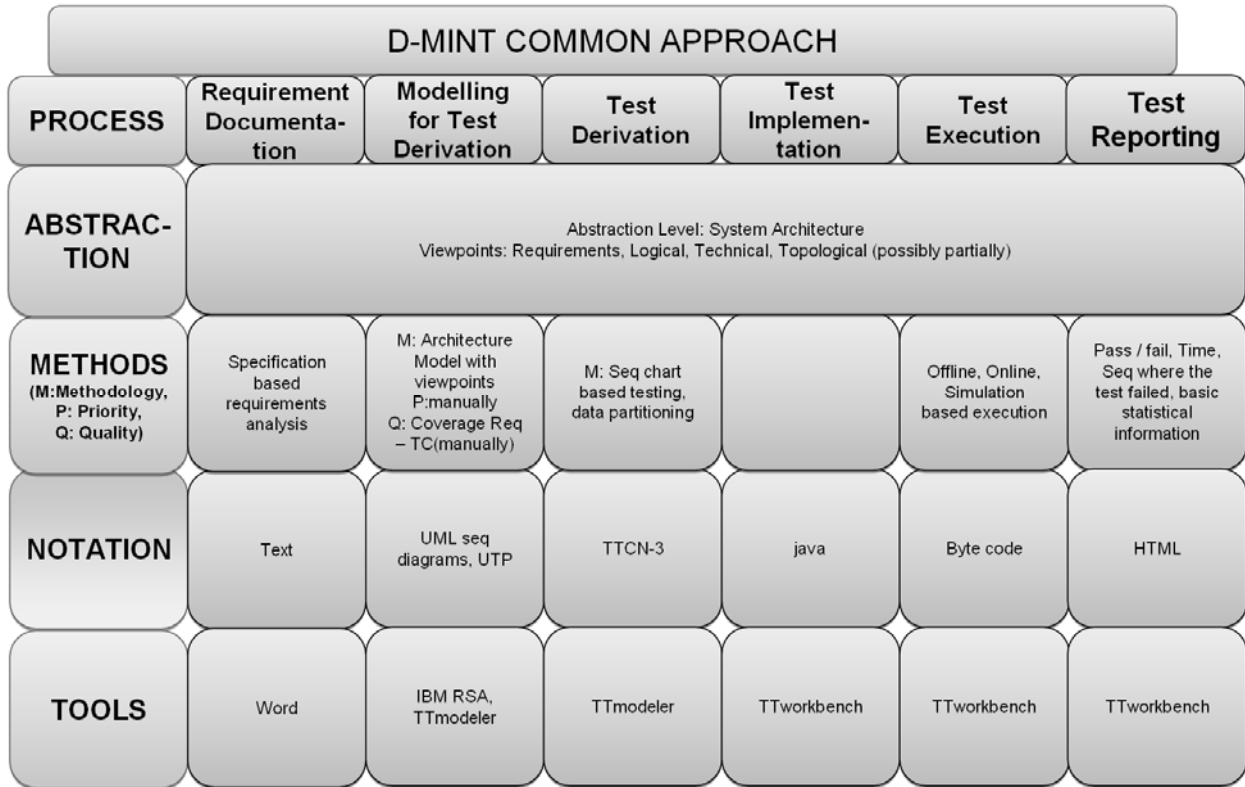
#### 4.5 IDEKO/SORALUCE CASE STUDY

In the IDEKO/Soraluce case study, *the system architecture* plays an important factor in test creation. The considered areas from the perspective of the common approach comprise *requirements, logical, and technical viewpoints*. The *topological viewpoint* is not included in the implementation of the SUT. Though, it might play a significant role in the testing environment of the machines and mechanical devices.

*Requirements specification* is performed based on the analysis of the textual documents written with word processing tools (e.g., Microsoft Word). Neither quality issues, nor the prioritization is regarded at this stage.

As a *model for test derivation* the system models in the form of UML sequence diagrams and use cases are used. Additionally, UML Testing Profile diagrams are applied for a direct test model specification. Then, the prioritization of the use cases is performed manually targeting at a full coverage of the requirements.

Hence, at this stage the definition of quality issues is being included. UML tool – IBM RSA is the most suitable tool selected for system modeling. TTmodeler aids the generation of TTCN-3 test cases.



**Figure 11 The IDEKO / Soralue Case Study**

The test derivation/generation is based on the analysis of UML sequence diagrams. For stimuli derivation data partitioning methods are applied. The generated test cases are specified in TTCN-3. The UTP model serves as an input to generate the TTCN-3 test cases in TTmodeler.

In terms of test implementation the test cases are transformed into Java source code and TTworkbench enables the compilation of TTCN-3 test cases into Java.

Finally, at Ideko test execution is performed offline in a simulation-based environment of TTman provided as a part of TTWorkbench. In the environment of Soralue, acceptance tests could be performed online directly in a machine tool.

Test reporting is performed in TTworkbench as well. Here, information about verdict, time, execution sequence, or errors is collected. Other supported features comprise test parameterization and retries depending on verdict. The reporting produces some basic statistical information about overall execution of the test suite. It allows for measurement and analysis of some quality indicators.

Execution data is stored in binary logging files and presented in table or graphical notation. The statistical reports are generated in basic HTML.

#### 4.6 NSN CASE STUDY

D-MINT COMMON APPROACH						
PROCESS	Requirement Documenta-tion	Modelling for Test Derivation	Test Derivation	Test Implemen-tation	Test Execution	Test Reporting
ABSTRAC-TION	Abstraction Level: System Architecture Viewpoints: Requirements, Logical (, Technical)					
METHODS (M:Methodology, P: Priority, Q: Quality)	Requirement Document; Structuring;	M: data, behavior, requirements modeling Q: Coverage: Req-SM, Modelvalidation;	M: State Model based TG; Scenario based Testing; data Partitioning; varios Coverage Criteria Q: Coverage Req-TC		Online, Offline	Log based analysis (Pass/Fail); Failed test tracking along requirements and model
NOTATION	Text (+Table), SysML	M: UML, QML, SysML Q: OCL	M: QML Q: Tool specific	M: EAST scripts	EAST scripts	Text, HTML
TOOLS	MagicDraw, Text	M: MagicDraw, Qtronic Modeller Q: MagicDraw	M: Qtronic, Python Q: Qtronic		Qtronic, EAST	Qtronic, Python, MagicDraw

Figure 12 The NSN Case Study

Nokia Siemens Networks [NSN] provides a telecommunication case study described in [DM08DM08]. The system under test of the case study is a MSC Server (MSS) which is a network element of 2G and 3G telecommunication networks. The case study focuses on specific features of the MSS specified by 3rd Generation Partnership Project (3GPP) [GPPGPPGPP]. The MSS provides the network side functionality of the mobility management (MM) feature.

In the case study the chosen *abstraction level* describe the network elements including the system under test and other network elements that communicate with the system under test. The applied viewpoints are i) *functional/requirements architectural view*, ii) *logical view*, and iii) *technical view*. In case of i) the requirements represent key tasks that the system under test should satisfy to perform the mobility management functionality. The network elements and interfaces between the network elements illustrate how the telecommunication network is structured (case ii). In case iii) configurations are used to express number of the required network elements by the tests.

The *requirements* are derived from specifications defined often using a natural language, for example, in English, with diagrams and tables. The requirements are structured using System Modeling Language (SysML) [SysMLSysML] using the MagicDraw tool and mapped to Unified Modeling Language (UML) [UMLUML] models created during the process as described in [AbbF09]. The requirements are propagated throughout the tool-chain to support requirement coverage analysis.

*Models for test derivation* are defined in UML using a structural work flow defined in [AbbJ09]. The models define the system under test with respect to behavior, data, and architectural configuration. The UML models are validated using Object Constraint Language (OCL) [OCL] rules with the help of the MagicDraw

tool. The UML models are transformed to Qtronic Modeling Language (QML) [Qtro09Qtro09] models before the tests are generated. The transformation phase exploits data, behavior, and configuration models as described in [P09P09P09P09].

Tests are *implemented* using a test generation feature of Conformiq Qtronic tool [Qtro09Qtro09]. Qtronic takes QML models as input and produces test cases using Nethawk EAST [Neth] scripting language. The *test generation* phase can be controlled using various coverage criteria options of Qtronic. After the test cases are generated, they are *executed* either using Nethawk EAST in offline mode or Conformiq Qtronic in on-line mode.

Nethawk EAST produces *test reports* in text format and message sequence charts whereas Conformiq Qtronic supports for example HTML format. The reports contain, in addition to regular verdict information (pass/fail), the requirement information created in requirement phase. The requirement information can be used to trace back the requirements associated to UML models for evaluating requirement coverage of executed tests and to speed up finding the potential sources of errors as explained in [AbbF09].

#### 4.7 TRIMEK CASE STUDY

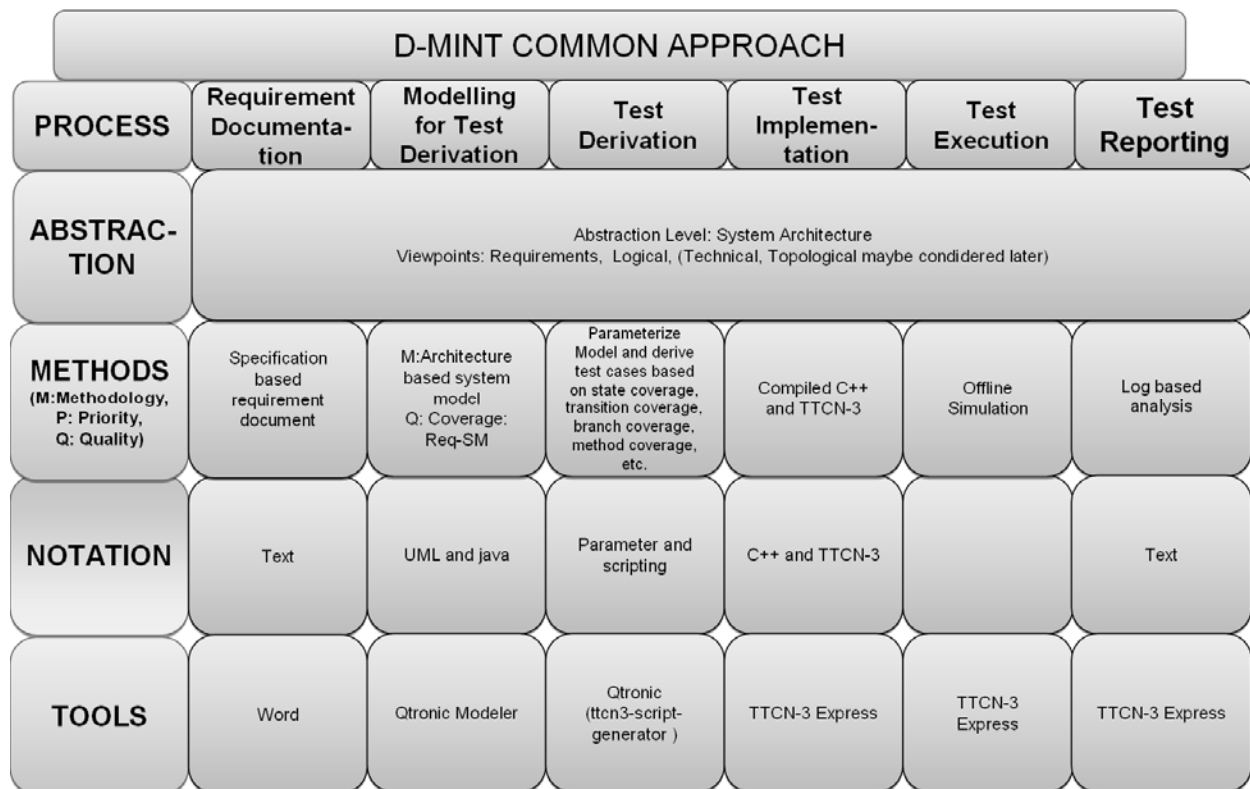



Figure 13 The Trimek Case Study

In the Trimek case study, the *requirements specification* is done in the form of description in various formats and is stored in a textual document with the help of Microsoft Word.

	White Paper: V 1.6	Page : 27 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

The considered *abstraction level* is related to system architecture, where *requirements and logical viewpoints* are considered. The technical and topological perspectives may be involved later when the formerly mentioned issues are handled in a reliable way.

As far as *modeling for test derivation* is concerned the abstract test cases can be retrieved from both the system model and test model designed in UML and enriched with Java behavior. Here, many combination options are possible. We can use direct way from the requirements through the system model to the test cases, or from the requirements through the test model to the test cases, or from the requirements through the system model to the test model and then to test cases. No matter what option is chosen coverage of the requirements in the models is targeted. The tool used at this stage is Qtronic Modeler.

The abstract test cases are gained from the models based on the model configuration. This phase of the process is called *test derivation*. The notation of the resulting tests is TTCN-3. With Qtronic we load the model in QML format that has been obtained in the previous steps. Then, some parameters, such as state coverage, transition coverage, branch coverage, or method coverage, etc. can be selected. With these parameters and the TTCN-3 script generator the test cases are produced.

Regarding the *test implementation*, executable TTCN-3 test cases are obtained and compiled C++ is used. The applied tool called TTCN-3 Express allows for execution of the tests. Though, in order to communicate the TTCN-3 test cases with the SUT the stimulus adapter is additionally created. It sends messages between TTCN-3 and the SUT. Then, also codec is build. This one is used to codify the messages.

The *test execution* follows in an offline manner. With the TTCN-3 Express we start the execution of the test cases through the SUT. The messages are sent to the SUT in order to validate the application.

After running the test cases, TTCN-3 Express creates a logging file with all information about the execution of the test cases, which corresponds to the *test reporting*. In order to verify the execution the log is reviewed.

## 5. ANNEX


### 5.1 REQUIREMENTS MANAGEMENT AND DOCUMENTATION

#### 5.1.1 Methods

In their vast majority, projects start from a document describing the requirements (the intended functionality and characteristics) of the system or component to be built. Such document may be in different formats starting from plain textual description, to structured text or tabular formats. The source of the requirements document is constituted by textual descriptions of the product, related standards and specifications, or user stories. The main purpose of this phase is to create a list of requirements for the product to document, structure and interrelate these requirements. Requirements can be specified on several levels of abstraction and are usually decomposed into different categories, depending on their nature, e.g., architectural, data, functional, non-functional, etc. The process of analyzing and structuring requirements is performed manually, based on the analyst's experience.

#### 5.1.2 Notations

The common approach in D-Mint has been to have some kind of textual description of the system from which a more structured requirements document has been created via the requirements analysis process. Several formats for requirements specification have been used. **Textual format** can be considered the

	White Paper: V 1.6	Page : 28 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

most basic. Sometimes the textual specifications are complemented with additional diagrams and tables following specific notations.

The **tabular notation** is a form in which the requirements are structured in tables with rows and sub-rows describing information about each requirement like, name, identification, source, priority, preconditions under which the requirement can be fulfilled, actions to be taken, conditions to be fulfilled after performing the actions, dependency on other requirements etc.

**Sequence based specification (SBS)** is a systematic procedure for developing (software) specifications that are consistent, complete and correct by construction [PP03]. The software system to be specified is considered as a black box, comprising a well-identified boundary with a defined set of input stimuli and output responses. The software requirements<sup>2</sup> serve as the basis for defining the boundary of this black box, as well as for identifying its stimuli and responses.

At the level of the black-box, the SBS process steps of requirements tagging, system boundary definition, sequence enumeration and canonical sequence analysis guarantee straightforward verification of completeness and consistency. Correctness is ensured by tracing the specified response for a stimulus sequence, back to the requirements. Consequently, ambiguous and/or missing requirements can be identified and appropriately corrected.


In the specification process, we

- (a) Tag individual requirements to facilitate traceability from the specifications.
- (b) Identify the system boundary and stimuli.
- (c) Methodically enumerate all possible sequences of stimuli in a strict order, i.e., stimulus sequences of length zero, one, two, etc.
- (d) Identify, from the requirements, the correct response(s) to be generated by the system for each stimulus sequence.
- (e) Apply reduction rules to reduce larger sequences to equivalent smaller *canonical* sequences; they are irreducible sequences that are not equivalent to a previous stimulus sequence. Canonical sequence analysis yields the state space for the system at the given abstraction level. By construction, canonical sequences are *disjoint*.

The enumeration terminates when all possible input sequences have been identified and their respective responses specified. Each input sequence can be considered as a usage scenario for the software. The SBS process requires the developers to consider all possible permutations and combinations of stimuli. Together with the requirements, SBS assists in distinguishing the impossible scenarios and erroneous usage from the correct and intended usage. Furthermore, as mentioned, irreducible sequences are also identified, forming the basis for precise specification of the software behavior.

**ETSI's standardized test specification.** In the first step, requirements are identified from one or more base specifications. These may be catalogued and published in a Requirements Catalogue. Then the Implementation Conformance or Interoperable Functions Statement (ICS/IFS) is constructed. These both are essentially high-level checklists for features and capabilities in a standard. The checklists are filled out by system vendors, according to which features they implement in their products. This information can help to determine if two implementations of the same standard have potential to interoperate. In the next step, one or more test purposes are specified for each testable identified requirement, either in English prose, or in the TPLan notation [TPLan]. A test purpose formulates (an aspect of) a requirement as a set of IUT pre-conditions, stimuli, and responses, and specifies test verdict criteria. The testability of a requirement is affected by the type of testing to be carried out, e.g., requirements related to error management cannot be assessed using interoperability tests because many error conditions cannot be triggered by conforming implementations. After the definition of test purposes, an informal test description

<sup>2</sup> The distinction between requirements and specifications is especially important: requirements refer to the state of the *environment* as it should be modified by the software or system, once it has been correctly constructed, whereas specifications refer to the desired behavior of the *software* at its interface to the environment.

	White Paper: V 1.6	Page : 29 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

can be specified in English prose, as tables, or as message sequences covering usually one, but sometimes multiple test purposes. Test descriptions extend test purposes by providing more detailed information of preambles and postambles. Test descriptions are by definition not executable. The preambles and postambles in test descriptions are not conceptually the focus of testing, even though they in practice depend on identified requirements in the same way as test bodies do. Therefore, conventionally preambles tend to be specified to invoke behavior that is most likely to be correctly implemented. In addition, preambles tend to be reused across test descriptions as much as possible.

**System Modeling Language (SysML)** [SysML] is a graphical modeling language derived from the Unified Modeling Language [UML/UML]. SysML provides Requirement diagrams as a means for graphically specifying requirements. The SysML requirements diagram is similar to a UML class diagram, describing the requirements structure, their relations, and attributes. A requirement in SysML includes an *id* and a *textual description* of the requirement. It is also possible to add user-defined properties such as *verification method* or use pre-defined requirement *categories* like functional, interface or performance. The relationships between requirements in SysML can be specified using relationships like DeriveReq, Satisfy, Verify, Refine, Trace, and Copy.

### 5.1.3 Tools


For requirements specified in tabular format, any text-based tool can be used. Typically, requirements structured in this way are specified using Microsoft **WORD** or similar text editors. Other specialized tools for requirements structuring like Telelogic **DOORS** can be also used. Requirements structured with the SysML requirements diagram can be specified with any modeling tool that supports SysML diagrams like NoMagic's **MagicDraw** [MagicDraw].

**PreeVision** [Preevision] is a modeling tool in the EE (electrical/electronic)/Automotive Domain that has been applied within the Daimler / FOKUS case study. The tool is produced by Aquintos in cooperation with Daimler. It allows modeling the automotive EE-aspects of the system architecture under the different viewpoints from the requirements, through the logical view system blocks to the technical view ECU, Busses, Sensor, Actuator configurations. It also allows modeling the topological view which was not of concern within this case study. A significant property is that it allows to specify the requirements in a "Precondition, Event, Reaction" formalism and to define links between the artifacts of the different architectural views.

**jSEQ** [jSEQ] is a tool for constructing sequence-based specifications from requirements. The information provided and developed in jSEQ is stored in a set of linked tables, e.g. for storing system inputs, valid input sequences, and expected outputs. At this stage jSEQ is only used for capturing the relevant requirements for testing the system. During the construction of the SBS all elements in the other tables will be linked to original requirements.

## 5.2 MODELING FOR TEST DERIVATION

As mentioned in the introduction, two different approaches are followed in D-Mint for creating test specifications. The first approach focuses on creating a *system model* of the SUT that is a model describing the behavior of the system from an internal perspective to system borders. Such models are similar to those typically used during the development of the system. The second approach focuses on creating a *test model* of the SUT, which basically describes the behavior of the system from an external perspective to its system borders. The later is the "model" used traditionally by testers to create test cases.

	White Paper: V 1.6	Page : 30 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

Both types of models are created from the requirements and can be situated on different viewpoints of the architecture-driven MBT process (see Section 2) or at different abstraction levels. The models will be used for test derivation in the later phases of the MBT process.

## 5.2.1 From requirements to System Model

As systems are getting more complex, there is a need to describe systems from different points of views and perspectives. Dealing with complex system, one usually wants to model (describe) several perspectives of the SUT, like architecture, data, behavior, etc. such that the information contained in these models can be used for automated test generation using specialized test generation tools.

### 5.2.1.1 Methods


**Use of Architecture and Viewpoints.** The architectural model shows with what components the systems interact and how the system is connected to its environment. The system is specified using different viewpoints (as described in Section 2).

**The AAU's modeling for automatic test derivation [MATERA] approach** [AbbJ09] pursued in the NSN case study starts from the *textual requirements* of the system and incrementally builds a collection of models describing the system from different perspectives. By *textual requirements* we refer to the collection of stakeholder requirements, as well as additional documents like protocol specifications, standards, etc. The modeling phase captures several perspectives (*domain*, functionality, *behavior*, and *data*) of the system, at several abstraction levels, by spanning the Functional Requirements and Logical View layers described in Section 2.

In order to cut down on the complexity of the specifications, the SUT is specified from several *perspectives*, as follows: **Domain modeling** describes the environment of the SUT from the perspective of external entities, their inter-connectivity and the interfaces provided. **Data modeling** focuses on specifying the data exchanged between the SUT and its environments from the point of view of types and structure. **Behavior modeling** is used from modeling the internal behavior of the SUT by showing how the later interacts with the environment. It shows what stimuli the system reacts to and what action the system performs to that stimuli. Finite state machines and Markov chains are two popular techniques in MBT for modeling system behavior. **Test Configuration modeling** is used to create specific test configurations by instantiating the entities in the application domain. **Requirements modeling** has textual specifications as input and describes how requirements are structured and related to each other. Requirements can be defined on several levels of abstraction and are usually decomposed into different categories, depending on the nature of the requirement, e.g., architectural, data, functional.

Each perspective is initially specified on the functional level via feature and requirements models, and it is subsequently refined on the logical model. There are both horizontal (between the perspectives on the same level), and vertical (refinements) relationships, respectively among the specification artifacts in this process, as it will be illustrated throughout this section. UML is used as a specification language for system modeling. Additionally, the *requirements diagrams* of SysML are used to capture the requirements of the system in a graphical manner.

The resulting set of models is transformed into the input format used by the Conformiq Qtronic tool which is used for automated test derivation. Test logs are collected and analyzed from both online and offline testing modes and the failed test cases are tracked back to the UML models from which they originated.

	<p>White Paper: V 1.6</p>	<p>Page : 31 of 51</p>
		<p>Version: 1.6 Date : 06/11/2009</p>
		<p>Status : Final/Released Confid : Public</p>

## Quality Evaluation

In the **MATERA** approach, there are two ways to check the quality of the resulting models. Firstly, the UML models are validated according to a set of custom validation rules before proceeding to test derivation. Secondly, by analyzing the test logs one can evaluate the quality of the generated tests against the models by observing how well the requirements have been covered.

**Model Validation.** Humans tend to make mistakes and forget things. Therefore, in modeling it is necessary to validate the models before using the models to i.e., automatically generate code or test cases. Model validation is performed to ensure that the models are syntactically correct and are free from errors. By validating the system models, one can be assured that the models are ready for test generation and that common modeling mistakes and errors are not propagated further to later steps in the testing process, where error detection is more costly. The validation will check the models for either consistency, correctness, completeness, or all of these. The idea behind consistency validation is to check for contradictions in the model. Correctness checks that the models conform to the modeling language, whereas completeness will check that all essential fields have been filled and that the information is correct. Validating models can be considered "best-practice" in modeling, since one is assured that i.e., relevant information is present or dependencies between elements are correct. Besides validation, by using a modeling convention the quality of the models can be significantly improved.


At UML level, this is achieved by writing OCL rules that applies for different kinds of UML elements. The models are later analyzed (validated) against the specified rules. The outcome of the validation is a notification saying which model element was violated by which rule. The rules are often arranged into different categories depending on the nature of the rule, e.g., domain specific, language specific, platform specific.

**Comparing Requirements Coverage vs. System Model** is the means to analyze what part of the model "implement" a specific requirement. It is also the means to check that all requirements specified in the requirements model are traced to elements in the system model. In this sense, model validation ensures that no requirement is overlooked and that all requirements are traceable. Since the system model is derived from requirements and is going to be used for test generation, having high traceability of requirements facilitates the process of tracing back the requirements from tests to the system model when tests fail.

## Prioritization

System requirements may be changed along system development due to reduction, extensions, adaptations or other modifications. Hence, it is important to identify those test cases in the test base that are affected by these changes imposing a redefinition of the corresponding test models. A new algorithm has been proposed in [H09] that support test case priorities based on weights and (feature) potentials (e.g. occurrence, risk, severity indicators). This enables an overall reduction in the magnitude of testing efforts, while minimizing resources needed for test execution. The input parameter values for the calculations can be assigned to model elements that are used for test case derivation, i.e. the system or test model.

The algorithm for the calculation of test case priorities is based on the sum of weight and potential values that have been assigned to the conditions and events in a model that represents the set of test cases in a test campaign. Within the first step all weight values of all possible tests (conditions) have to be calculated by multiplying all single weight values that belong to a single test in the model. In a second step of the algorithm a factor "F" will be calculated for each test that intends to reflect the relevance of a test case in addition to its test case weight. This factor "F" identifies the portion of condition of a test case together with a consideration of the potential values assigned to test in comparison to the total sum of uncovered conditions and test potentials. Next step is the multiplication of condition weight values and the

	White Paper: V 1.6	Page : 32 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

factor “F” of each test case. The priorities of the test cases are according to the resulting calculated values: the test case with the highest value has the highest priority, etc.

### 5.2.1.2 Notations


**Unified Modeling Language (UML)** [UML] is a standardized modeling language used in the field of software development. UML is a graphical language for visualizing, specifying, and constructing models and offers a standard way to build system models. UML offers a way to create graphical abstract models of specific systems and it allows software developers to concentrate more on design and architecture of creating a system rather than just code implementation.

UML offers 13 different diagram types; each diagram type describes the system in their own way. These diagrams are divided into three different views of the system, a structural view, a behavioral view, and a functional requirements view. The structural view expresses the static structure of the system using objects, attributes, operations, and relationships. This is achieved with class diagrams and structure diagrams. The behavioral view on the other side expresses the dynamic behavior of the system showing interaction between objects and changes to the internal states of these objects. This view includes state machine diagrams, activity diagrams and sequence diagrams. The requirements view emphasizes the functional requirements of the system from the user's point of view, using use case diagrams to describe requirements. All diagrams are divided into three main categories: Six diagram types represent structure, three represent types of behavior, and four represent different types of interactions

**The Object Constraint Language (OCL)** [OCL] is a formal language developed by OMG to describe constraints on UML model elements and is nowadays a part of the UML specification. The notation used in OCL is similar to object-oriented languages and thus most expressions can be read from left to right. OCL also provides the usage of variables and operations which can be used in various combinations to 13 build expressions. Even though OCL allows usage variables and operations, the language is not a programming language and therefore cannot be used to invoke operations or alter information in the model. The language is a pure specification language without side effects. This means that the state of the system is not altered when an OCL expression is evaluated.

OCL expressions can be used for a number of different proposes with UML models. In general, an OCL expression can used to specify constraints on every model element specified in the UML specification. However, OCL is typically used to specify invariant constraints on model elements such as classes, and to describe pre- and post conditions for operations and methods

**System Modeling Language (SysML)** [SysML] is a graphical modeling language for system engineering developed by the OMG. SysML is basically a UML profile that represents a subset of the UML 2.0 with some extensions. SysML supports the specification, analysis, design, verification, and validation of systems. SysML is meant to be consistent and compatible with UML 2.0, and extends UML 2.0 when it is insufficient to meet certain goals. It supports the specification, analysis, design, verification, and validation of a broad range of systems and systems-of-systems. These systems may include hardware, software, information, processes, personnel, and facilities. SysML provides Requirement diagrams as a means for graphically specifying requirements. The SysML requirements diagram is similar to a UML class diagram, describing the requirements structure, their relations, and attributes. A requirement in SysML includes an *id* and a *textual description* of the requirement. It is also possible to add user-defined properties such as *verification method* or use pre-defined requirement *categories* like functional, interface or performance. The relationships between requirements in SysML can be specified using relationships like DeriveReq, Satisfy, Verify, Refine, Trace, and Copy.

	White Paper: V 1.6	Page : 33 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

### 5.2.1.3 Tools

**MagicDraw** [MagicDraw] is a software and system-modeling tool, developed by NoMagic, which offers support for analysis and design for object oriented systems and databases using the UML language. With MagicDraw's built in OCL interpreter it is possible to validate the models via validation constraints written in OCL. It is possible to extend MagicDraw via add-ons. MagicDraw offers support for e.g., SysML, and can import models from other UML tools via the use of plug-ins. Elements from different modeling languages (e.g., UML and SysML) can be linked to each other via graphical editors, facilitating thus traceability across models and languages.

MagicDraw also offers support for code generation for languages such as Java, C++, and C#, as well as database schema modeling and reverse engineering facilities. Using MagicDraw's teamwork server, multiple people can work on the same model simultaneously. With MagicDraw's automatic report generator, one can produce design and requirements documents in PDF, HTML, and RTF format.


**MATERA** is a plug-in for MagicDraw developed at the Åbo Akademi University, Finland, that integrates different the steps of the MATERA approach with the graphical interface of the tool. The plug-in provides facilities to validate the models created before proceeding to the automatic generation of the model used for test generation. From the same environments, one can invoke the function that collects the statistics of the test results, tracks back the test failed test cases to models, and displays in the graphical editors of MD those elements that are linked to the failed requirements.

In **PreeVision** [Preevision], the requirements and their parts can be linked to the functional blocks in the logical view that realize them. Those functional blocks in the logical view in turn can be linked to the technical view blocks (sensors, ECUs, actuators) they are to be implemented on. The same holds for the communication ports and communication between the architectural components.

**StarUML** [StarUML] is an open source project to develop fast, flexible, extensible, and freely-available UML/MDA platform running on Win32 platform. The goal of the StarUML project is to build a software modeling tool and also platform that is a compelling replacement of commercial UML tools such as Rational Rose, Together and so on. It is implementing UML2. It supports code generation and reverse engineering within a Java, C++, and C# profile.

**IBM Rational Software Architect (RSA)** [RSA] is one of the UML2 tools in the IBM modeling tool collection. It is the successor of the famous Rational Modeling tools, well known from the time of UML1. RSA is an eclipse based, commercial, integrated modeling and development platform. It allows modeling and graphical editing across a variety of different domains e.g. UML2, Java, Web, technical infrastructure etc. In addition to UML 2 and domain specific modeling, it supports model transformations. An open API allows the customization and extension of the modeling environment.

**Enterprise Architect (EA)** [EA] is a UML tool built upon the UML2.1 specification. It supports the UML profiling mechanism and also offers a realization of the SysML specification. It allows modeling and graphical editing of models across a variety of different domains. It supports Model Driven Architecture transformations using transformation templates. It is extendable and adaptable to specific technologies and frameworks using an EA specific MDG technology.

	White Paper: V 1.6	Page : 34 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

## 5.2.2 From requirements to Test Model

### 5.2.2.1 Methods

**Message Sequence Charts (MSC)**, or sequence diagrams, are primarily used to describe the interactions between objects in a sequential order, as well as for describing the behavior of a use case, by showing the messages (and their parameters) that are passed between objects for a given use case.

**The TPT approach [TPT].** Test case generation using *Time Partitioning Testing* (TPT) is based on abstract test models or usage models. A TPT test/usage model is a “superset state machine” that can be seen as the union of all relevant test cases. With TPT each individual test case is not just a constant sequence of test steps but a full state machine that is part of the superset state machine modeled with TPT. TPT test models are hybrid state machines that allow the stimulation and observation of discrete and continuous input and output of the SUT. This concept allows the definition of powerful reactive tests (also known as closed-loop tests) where the test case behavior may depend on the system behavior at runtime (online).


Test cases with TPT are generated offline either manually or automatically using combination rules (test pattern rules). Test generation is the process of selecting, compiling, linking (with additional resources) and mapping test models to the actual SUT.

TPT test cases describe the abstract semantics of the desired test behavior. The mapping from the test models to the actual SUT is done by means of so called platform adapters that define the mapping rules between the test model and the SUT, i.e., how to represent test cases in the desired test environment and how to stimulate and observe the concrete SUT interface. With that the TPT test models can be shared and reused between different SUT implementations. This feature is highly relevant in the embedded domain where the system development goes through a couple of development phases – from initial models through the implemented software to the final embedded device.

In the domain of embedded control systems, *system parameters* are crucial to be considered when testing. System parameters are configuration settings that allow an embedded system to be used in different environments. System parameters specify exactly the environment specific setup. In this sense an embedded control system constitutes a system family that has to be tested in a broader variety of scenarios. Usually the following categories of system parameters exist:

- **Enabling parameters:** Parameters that signal the existence/absence of physical or SW-features in the environment of the SUT. In dependence of those parameter settings some system algorithms/features etc. behave different.
- **Setup parameters:** Parameters that can be tuned to specific physical (e.g., electrical, mechanical) or legal ranges. Those parameters are usually used to adjust the system functionality to perfectly interact with the physical environment or to meet legal constraints of particular countries.
- **Adoptable parameters:** Parameters that can be tuned at runtime to adjust the system behavior automatically depending on the online system observation functions and the expected, built-in obsolescence of mechanical parts.

When testing systems with parameters the test models must support parameter definitions and parameter variation. Since parameter setup can be rather complex in practice and there are many dependencies between the chosen parameter settings, parameters should be defined not only in the test model only, but in both the test model *and* the SUT. This feature requires a handshake protocol of parameter settings between the test model and the SUT.

	White Paper: V 1.6	Page : 35 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

Before executing a test case parameter settings are requested from the SUT by TPT. Usually these parameters are defined in the simulation or test environment of the SUT – at least for a system parameter subset. As soon as TPT receives the parameter settings it will be checked which of these parameters are defined in the test model too and which parameters are not defined in the test environment in the first place. These parameters have to be overridden in the system setup. For that purpose TPT sends the parameter bindings to the SUT. After confirmation that parameter settings have been adjusted TPT transforms the abstract test cases together with the parameter bindings into executable test cases. Finally these generated test cases are downloaded and executed in the actual test environment.


**The REDUCE approach.** In the REDUCE approach [BAU09], we use model-based statistical testing (MBST) to construct the generic test models and to automatically generate test cases from the concrete test models. In the test modeling stage the original system requirements are systematically inspected by the sequence-based specification method to determine the system boundary and the stimuli and responses of the test object. The next step is the sequence enumeration which aims at systematically writing down each possible stimulus sequence, starting with sequences of length one. A sequence of stimuli represents one history of the usage of the system under test. Additionally, every sequence is compared to previously analyzed sequences w.r.t. equivalence. Two sequences are equivalent if their responses to future stimuli are identical. Illegal and equivalent sequences are not extended in the ongoing sequence enumeration. The enumeration stops if all sequences are illegal or reduced to equivalent sequences.

**The Pattern-Oriented Model-Driven Test Engineering approach** [AV01]. This approach consists in automatically generating test artifacts (e.g. executable test scripts or skeletons thereof) based on dedicated test models designed following some previously defined and well-documented patterns. The concept of test patterns has been around for a while. However, with the great number of different approaches on testing with regard to scope (e.g., unit testing, sub-system-testing, system-testing), method (i.e., white-box, grey-box, black-box-testing) or purpose (i.e., conformance, interoperability, performance, reliability), the notion of test patterns can be explained in many ways. Patterns can be defined at different levels of abstractions, ranging from high-level non-functional patterns describing good-practices for a certain domain to more functional (instantiable) ones like object-oriented design pattern. The FOKUS concept of test patterns aim at embodying the knowledge gathered by test designers in such a way that, that knowledge could be used to drive intelligent test design and test generation.

Test patterns cover all aspects of test modeling, i.e., test architecture, test data and test behavior modeling. Therefore, pattern driven test generation allows a more focused generation of test assets (test configurations, test data, test behavior), instead of the brute-force approaches based on emulating elements of the SUT. While existing test generation approaches consist in combining all possible input-output-state triplets for a given SUT based on its model, pattern-based test generation uses additional information stemming from domain-specific or well established and documented good practices to reduce the amount of test cases generated to those that have a higher potential in assessing the SUT's correctness.

Patterns are best expressed at a conceptual level of abstraction, using MDA approaches. Therefore, it was a natural step to consider existing test modeling approaches for the purpose of expressing "instantiable" test patterns for integration in the test development process. The Unified Test Modeling Language (UTML) is a DSL for test design defined by FOKUS that embodies concepts of pattern-oriented model-driven test engineering. A prototype implementation of a UTML tool chain has been developed in the D-Mint project and successfully applied on a case study featuring interoperability testing for implementations of the IP Multimedia Subsystem (IMS).

**Quality Evaluation.** As for every model-related testing approach, the quality of the test model is a critical aspect of pattern-oriented MDT. Therefore, techniques for evaluating and improving the quality of test models were considered right away. Those techniques cover both syntactical analysis of the model based

	White Paper: V 1.6	Page : 36 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

on the UTML meta-model and semantic analysis using OCL constraints. This ensures that the resulting test models are valid from an early design phase and that flaws in the test models are not propagated down to the generated test artifacts.

**Prioritization.** Prioritization is another important aspect of test modeling to facilitate decisions on test implementation and test execution at a later stage. The UTML notation supports manual prioritization of test objectives based on criteria that can be calculated automatically using system models as described in [KTH05] or assigned arbitrarily by test project managers depending on proprietary criteria.

### 5.2.2.2 Notations

**UML Sequence diagrams** are the adoption of message-sequence charts (MSC), often used for modeling telecommunication protocols, in the UML 2. They show the interaction in terms of temporarily ordered method invocations with their input and return parameters. They allow to model loops and concurrency. Sequence diagrams can be used to describe communication scenarios in the context of use cases, to describe message exchange scenarios in the context of communication protocols as well as test scenarios. Sequence diagrams can also be used to describe system behavior and simulate it by “executing” those sequence diagrams in a model simulation engine.

**The UML Testing Profile (UTP)** [U2TP04]. UML is widely used as a modeling language. It offers a mechanism to extend and restrict it, called profiling. UTP is such a profile. The profile specializes UML for the domain of testing. It offers modeling concepts to describe the

- *test architecture*, the structural aspects of the test situation,
- *the test context*, allowing to group test cases,
- *the test configuration*, showing the structural configuration of test components and SUT and the communication between them,
- *the SUT*, one or more objects forming the system to be tested,
- *the test components*, objects within the test system that communicate with the SUT to realize the test behavior,
- *the test case*, describing the pre and post conditions; events and event sequences; input parameters; expected and observed results; some sort of validation, that compares expected and observed results; a verdict.


UTP allows specifying the structural as well as the behavioral aspects of tests. Additional time related concepts are introduced to supplement the already available simple timing concepts of UML2.

Some additional concepts like the *scheduler*, *arbiter*, *test log* etc. are defined for advanced purposes.

Due to the fact, that UTP is defined as a UML Profile, all UML tools that support the UML profiling mechanism should be able to support the UML Testing Profile.

The UML-2 Test Profile (U2TP or UTP) is the standard notation defined by the OMG for UML-based model driven testing. It defines a set of extensions to the UML 2.0 standard to enable test modeling with UML. The extensions defined in the UTP are grouped in 4 categories:

- Test architecture
- Test behavior
- Test data
- Time concepts

	White Paper: V 1.6	Page : 37 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

Those four categories of extensions provided by the UTP match perfectly with the categories of test-patterns FOKUS defined in previous works, namely test configuration, test behavior and test data patterns. Obviously, the only difference was that FOKUS integrated time concepts to behavior patterns instead of defining them as a separate group of patterns. However, while the UTP in its current version provides most of the basic concepts required for pattern driven test modeling, these were not sufficient and hence some new concepts were needed, along with a refinement of some existing concepts. The extended UTP including those new concepts was then modeled as a meta-model for a notation FOKUS called PTML (Pattern driven Test Modeling Language).

The **PTML** metamodel implements the concepts of the pattern-driven test modeling. The elements of that metamodel are presented below, along with the corresponding UTP concepts, where applicable. The main packages of the PTML model reflect four main groups of concepts. Additionally to the three groups of concepts already contained in the UTP, i.e., the *test\_data*, *test\_behavior* and *test\_configuration* packages, the PTML introduces two new groups of concepts as packages: the *test\_objectives* package and the *test\_patterns* package.


The *test\_objectives* package contains all concepts related to the modeling of test objectives. The advantage of modeling the test objectives and the other aspects of testing using the same notation is that test cases can be linked to test objectives to ensure a tracing back to those test objectives during test execution. Furthermore, provided a connection is made between user and system requirements and test objectives, executed test cases could even be traced back to those to evaluate coverage.

The *test\_patterns* package defines concepts allowing the definition of test patterns, i.e., generic solutions for recurring testing problems for a specific domain or organization. An example of such a pattern could be one defining the composition of a test case as a preamble-test body-postamble triplet, or as a test body-postamble pair etc. Based on such test patterns wizards could be generated to assist the test designer in defining test cases following that pattern. An automatic validation of defined test cases could be provided as well.

**Markov chains** are state machines annotated with transition probabilities. In model-based statistical testing Markov chains are used to describe the expected usage of the test object (also called usage model or test model [Pro05]). The Markov probability law requires that the future is based on the present and not on the past. Although the Markov law often does not apply to states within a hardware-software system, it almost always applies to states of use of a system. The user typically relies only on the present and not the past. Mealy machine construction complies with the property. The usage is reflected in the transition probabilities. One may apply several usage profiles for a single usage model (e.g. climate-specific or country-specific profiles). For risk-based testing the model transitions may be annotated with risk assessments. Risk values can incorporate probability or frequency and the risk impact. The transition probability influences the statistical analysis of the test model and the test case generation. Data for probability distributions is sometimes available from domain experts or monitoring data from similar systems, partially or completely, for usage environments. When data is not available, uniform probabilities are used. All outcomes are regarded as equally likely.

### 5.2.2.3 Tools

The **TTmodeler** tool [[TTM09] integrates the model-based system and test design and creates a direct, automated connection to the TTworkbench test development and execution environment. The UML Testing Profile (UTP) defines additional model elements for UML, which allow the definition of specifications of test objectives, test procedures and test valuations for systems and system components. UTP is a vital component for the alignment of system and test development methods. It contributes to the concept of model-centric systems design: System models and test models can be developed and aligned in all system development phases. The benefits resulting from application of TTmodeler are the alignment of system design and test design, model-driven system development approach, direct automated execution of test models, standards-based testing with TTCN-3 and UML testing profile.

	White Paper: V 1.6	Page : 38 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

This model-driven test development approach starts by enriching a system model described with UML by test directives on the basis of the UTP. Throughout this activity the validation of the test model is covered by the UML tool support and by the rules added to it by the additional UML Test Profile. In the second step, the test model is exported to a standardized UML model, internally represented by the MDT/UML2 UML metamodel. This additional step is performed in order to be able to integrate TTmodeler in a heterogeneous environment of UML design tools, each having different architectures and interchange formats. By applying the mapping rules, the intermediate UML test model is transformed into the internal TTCN-3 metamodel of TTworkbench.

**Piketec TPT** [TPT] is a model-based test tool used in the testing of embedded control systems. It supports full automation testing in a real-time context and can use continuous as well as discrete signals. Test cases are created using graphical test models based on specialized hybrid state machines. Thanks to the built-in platform independence, features such as parallel and hierarchical test flow, conditional branching as well as support for continuous and discrete signals allow the construction of complex test cases. TPT can be used universally from Model-test (MiL), through Software-test (SiL) to Hardware-in-the-Loop-Test (HiL). For the testing of complex systems TPT offers a powerful approach for systematic test-case definition which guarantees easy interaction and readability even with a larger number of tests. This is handled by a specialized variation concept: All test cases are represented as instances of a single state machine. The states and transitions in this state machine allow the definition of multiple variants. The test cases differ in terms of the variant selection and, therefore, combinations of variants constitute the actual test cases.

**jSEQ** [JSEQ] is a tool for constructing sequence-based specifications from requirements. The information provided and developed in jSEQ is stored in a set of linked tables, e.g. for storing system inputs, valid input sequences, and expected outputs. The sequence-based specification is converted to a state-based model which describes the valid input-output trajectories of the test object. The state-based model is annotated with statistical information about the importance (e.g. frequency of use or criticality) of model transitions.


For the automated model-based testing the tool **JUMBL** [JUMBL] is used. JUMBL is a command line application developed by the SQRL group at the University of Tennessee. JUMBL supports the following steps: statistical analysis of the test model, automated test case generation, statistical evaluation of the test results. The test model analysis is used to assure the model plausibility by applying simple model checking techniques and statistical model analysis. At first, the model structure is checked i.e. that all states are connected to the Start and Exit state and that unique names are used for states and the outgoing arcs of every state. Furthermore, the test model can be statistically analyzed, e.g. the test model complexity and the probability distribution implemented in the model.

## 5.3 TEST DERIVATION

### 5.3.1 Test Derivation From System Models

#### 5.3.1.1 Methods

**Scenario based Test Derivation.** The main goal in scenario based test derivation is to cover and test explicitly selected usage scenarios, as opposed to automated test generation where paths in a given behavioral model of the system are covered based on selected coverage criteria. The main benefit with

	White Paper: V 1.6	Page : 39 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

scenario-based testing is that it would facilitate the acceptance testing process, by allowing testers to validate customer requirements via automatic test generation for specific usage scenarios.

**Coverage criteria based Test Derivation.** The main goal in coverage criteria based derivation is to generate test cases based on coverage criteria, i.e., requirements coverage, transition coverage, state coverage, that are specified before starting test generation. The coverage criteria are “guidelines” that tell how the test cases are to be generated and how they cover the system model. Test generation ends when one has reached the desired level of coverage (normally 100%).

### 5.3.1.2 Notations


The **Qtronic Modeling Language (QML)** [Qtro09] is a combination of UML state charts and textual source files with a Java-like syntax, and is used for describing different aspects of the SUT like input/output ports and messages, message structure, behavior, etc. The textual source files are an extended notation of Java and are used for describing data types, constants, classes and their methods. The extended notation of Java provides support for value-type records, macros, non-deterministic programming, and requirements traceability.

### 5.3.1.3 Tools

The **Conformiq Qtronic** [Qtro09] is a model-based test design solution. It supports offline test case generation based on models in the industry standard programming and modeling languages (e.g., Java, C#, and UML) [Conf]. It let derive tests automatically from system models, i.e., artifacts that represent and model the desired behavior of the system under test. It uses semantics-driven methods for generating test suites. That is, test generation is guided by deep state space analysis of the behavior implied by the model, instead of being based on syntactic analysis or simple heuristics.

The diagrams can be drawn in various tools that Conformiq Qtronic works with, such as **Conformiq Modeler**, **Enterprise Architect** or **IBM Rational Software Developer**. It is also possible to create models completely textually, i.e. all the diagram types are optional. **Conformiq Modeler** is a free lightweight modelling tool from Conformiq that can be used for creating the system model UML state machine.

Given a system model, the tool automatically identifies a number of test cases that together cover the requirements selected for test generation. Appropriate test input data as well as the correct expected output is automatically calculated and generated by the tool without any further input from the user. For this, Conformiq Qtronic uses *semantics-driven methods* for generating test suites, which means that test generation is guided by deep state space analysis of the behavior implied by the model, instead of being based on syntactic analysis or simple heuristics. Qtronic uses *model driven coverage criteria* to select a set of test cases to form a good test suite. The *coverage goals* are used to guide Qtronic to look for certain behaviors from models or to enable certain behaviors: a test case covers a certain coverage goal if execution of the test against the model itself would cause the goal to be exercised. Then Qtronic uses its capability to simulate the system model symbolically to construct test cases and at the same time it maps the test cases to the different test goals induced by the coverage settings. It then selects from the test cases it has constructed a set that covers all the found test goals using a minimal cost test suite. This ensures that the suite is reasonably small and compact but at the same time the individual test cases remain relatively short, which eases test execution and debugging. In addition to this, Conformiq Qtronic also prefers to cover all test goals as early as possible, i.e. after as few messages as possible, providing better separation of concerns between test cases.

	White Paper: V 1.6	Page : 40 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

Hence, Qtronic automates the selection of tests and test data, and the computation of the expected results. It is a complementary rather than a competing product to most of the existing test automation solutions. The tool uses a model of the system under test as a source, generates comprehensive test sets using selectable test design heuristics and writes the test sets for example into a database. The test sets can be later executed independently of the tool [Qtro09].


Qtronic supports both online and offline test generation. The behavior of the SUT can be described either textually (in QML) or graphically as a mix of UML state machines and QML used as action language. Qtronic generates test cases by following paths through the system specification based on the selected coverage criteria.

Once the test cases are generated, the Qtronic user interface provides several different views allowing the user to do detailed analysis of the generated tests. Qtronic stores the generated test cases in persistence data storage and they are used as input to consecutive incremental test generation runs providing faster test generation. In order to meet the need of generating separate test scripts that can be stored to version control and executed independently afterwards against the system under test using an existing test harness and test execution environment, Qtronic provides the means of rendering automatically generated test cases in a variety of formats. Test rendering is carried out using scripting backends that are connected to Conformiq Qtronic via well-defined open API. Qtronic comes with a collection of scripting backends, but users can also create themselves scripting backends that generate desired output formats.

Elvior's **MOTES** [EMEM] is a test generator that generates TTCN-3 test cases from the IUT EFSM model. MOTES is used to generate functional test for the black-box testing of the SUT. The model for MOTES consists of EFSM, context variables, and port definitions. EFSM is used to model the behavior of the IUT. With MOTES third party UML CASE tools are used for drawing EFSMs. Currently MOTES can import state machines from Poseidon UML CASE tool. In the future MOTES will export state machines from any UML CASE tools that support export in XMI 2.1 format. EFSMs for MOTES are drawn as a flat UML state model without parallel and hierarchical states. UML state machines do not have formally specified action language for presenting guard conditions, input events and actions on the transitions. It is possible to generate tests only from the formal system model. Therefore a formal action language for MOTES purposes is developed for using on transitions of the EFSM. Context variables are the variables used in EFSM for storing state information in addition to control states of the EFSM. Port definitions define the input/output ports used in the model. Ports define the IUT interface towards its environment. Port definitions define the data type of the events that the ports accept. The context variables and ports are defined in TTCN-3.

Tests control part (or test sequences) is generated by MOTES automatically from the IUT model. Test data used by the test sequences should be prepared manually before the tests generation. Like context variables and port definitions also the test data are defined and instantiated in TTCN-3.

MOTES uses structural coverage criteria on the EFSM. The following options are possible: *selected states/transitions*, *all transitions*, *all states*, *all k-switches*. MOTES includes two test generation engines: (1) *model checking engine* and (2) *reactive planning tester engine*. The *model checking engine* is used for generating test cases for the deterministic IUT. The engine utilizes the Uppaal Cora model checker for finding abstract test sequences over the EFSM. The *reactive planning tester engine* does not generate test cases but it generates a reactive planning tester that in test execution time generates tests on-the-fly. The reactive planning tester is targeted for generating tests for nondeterministic but it can be used for generating tests for deterministic IUT also. Reactive planning tester engine uses reachability analysis in synthesizing the reactive planning tester. The reactive planning tester is generated in form of TTCN-3 script. The intelligence of the reactive planning tester is encoded as gain functions into the relationship between test coverage items. In on-the-fly testing phase the reactive planning tester has to decide which

	White Paper: V 1.6	Page : 41 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

move from the possible ones to take. For each possible move it calculates the gain function and takes the move with maximal gains. The gain function gives higher value if the next move leads faster to bigger amount of next coverage items.

After the previous steps are completed the user has to press a “G” button and the test generator does the rest. The TTCN-3 test cases or reactive planning tester are generated under the current *resource set*. The generated TTCN-3 files can be imported to any TTCN-3 test tool and to run them against the IUT. MOTES is used for tests generation in ELIKO street lighting controller case study.

**PreeVision** [Preevision] offers a facility model queries and report writing, that can be used and in the project was used to derive test descriptions in textual format (Open Office documents) from the system architecture model. This can also be seen as a model-to-text transformation. Alternatively, a model-to-model transformation facility, the tool PreeVision is build on, could have been used to derive a test model from the system architecture model. That would then have been used to derive divers textual test descriptions from it using a model to text transformation facility. This way would have been the more elegant and flexible way to derive tests from the system architecture, but could not have been used for the case study since it had not been released and documented for public use.

## 5.3.2 Test Derivation from Test Models

### 5.3.2.1 Methods


**Sequence Chart based Test Derivation.** According to the degree of abstraction and completeness test models already may include behavioral sequences representing the test case scenarios. Depending on the used notation for the abstract test cases some mapping is needed to translate the test scenario towards the target test implementation language. For example, in [DAI06] the differences between UTP and TTCN-3 have been addressed and some mapping rules for test behavior had been introduced: A significant difference that has been mentioned is due to the fact that UTP allows more description technique, since behavior diagrams may provide either a local (e.g. state machines) or a global view (e.g. interaction or activity diagrams on the overall system) of an object. While the mapping of some behavioral statement (stimuli, observation, coordination) could be realized straight forward this is not the case for e.g. default and verdict handling. Special attention is also needed test data due to more powerful language features like data pools, data partitions or data selectors.

**Data Partitioning.** Special emphasis is given to the engineering of test data. Usually approaches introduce classification or partitions and use representatives instead of all or a subset of values from the divided input space. Partitioning the range of inputs into groups of equivalent test data aims at avoiding redundant testing and improving the test efficiency and coverage. The situation is becoming even more complex when hybrid systems are considered since their signals vary continuously in value and time. For explanation reasons let us focus on the value domain.

Practitioners often define equivalent classes intuitively, relying primarily on case studies. The success of this method depends on the tester’s experience and his subjective judgments. Another option is to specify the equivalence [B03] based on the requirements. This approach depends on the fact of whether the specification provides sufficient details from which the equivalence classes and boundaries could be derived. At least three different methods can be used to choose the representatives of the equivalence class, namely random testing, mean value testing, and boundary testing.

### Statistical generation and selection of test cases

Test cases may be automatically generated from Markov chain usage models as model paths. Usage models contain a particular state for the start and a particular state for the ending of test cases. Usage

	<p>White Paper: V 1.6</p>	<p>Page : 42 of 51</p>
		<p>Version: 1.6 Date : 06/11/2009</p>
		<p>Status : Final/Released Confid : Public</p>

models may be annotated with so-called usage profiles, a probability distribution of the expected inputs at all usage states. In general, usage profiles describe the frequency of inputs. If additional data is available other profile like risk and costs could be also applied. Costs describe to the expected costs for executing and evaluating a particular transition. Risk corresponds to the expected costs when a transition causes a failure. Risk is calculated from the frequency (probability) and the impact or damage (amount of money). The statistical testing method includes various ways of automated test case generation, which means various strategies to walk through the usage model:

- generating the minimum coverage set for a usage model, covering each state and transition in the minimum number of test cases and test steps
- generating random tests based on a probability distribution
- generating tests in order by weight (most or least probable paths through the model)

After the generation test cases can be assessed for further prioritization. A test case is a sequence of test steps. Each step corresponds to a transition of the test model. The annotation of transitions traversed by approaches can be used to process the transition annotations:

- product (multiplying the transition weights)
- sum
- average value (arithmetical mean)
- median value

For the assessment of the probability of a test case the probability values of the transitions are multiplied. Since the probability is a value between 0 and 1, short test cases get higher probabilities. If costs for the test execution are annotated in the model, the resulting test cases can be assessed by their cost (sum of the transition cost). In this example the cost of a test case corresponds to its length. For the assessment of risk coverage the same approach (sum) may be used. The same test case can be assessed by several weights (e.g. probability, test execution costs, risk coverage).


### 5.3.2.2 Notations

**Testing and Test Control Notation (3rd edition) (TTCN-3) [ETSI07]** is widely accepted as a standard language for test system development in the telecommunication and data communication area. It covers concepts suitable to all types of distributed system testing.

TTCN-3 is a test specification and implementation language to define test procedures for black-box/grey-box testing of distributed systems. Stimuli are given to the system under test (SUT). Its reactions are observed and compared with the expected ones. Based on this comparison, the subsequent test behavior is determined or the test verdict is assigned. If expected and observed responses differ, then a fault has been discovered which is indicated by a test verdict fail. A successful test is indicated by a test verdict pass.

The language allows for the description of complex distributed test behavior in terms of sequences, alternatives, loops and parallel stimuli or responses. Stimuli and responses are exchanged at the interfaces of the system under test, which are defined as a collection of ports being either message-based for asynchronous communication or signature-based for synchronous communication. The test system can use any number of test components to perform test procedures in parallel. The interfaces of the test components are described as ports.

TTCN-3 is a modular language and has a similar look to a typical programming language. Additionally to the typical programming constructs, it contains all the important features necessary to specify test campaigns for functional, conformance, interoperability, load and scalability tests. These are test verdicts, matching mechanisms to compare the reactions of the SUT with the expected range of values, timer handling, distributed test components, ability to specify encoding information, synchronous and asynchronous communication and monitoring.

	White Paper: V 1.6	Page : 43 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

TTCN-3 test specification consists of four main parts:

- type definitions for test data structures
- templates definitions for concrete test data
- function and test case definitions for test behavior
- control definitions for the execution of test cases

Historically, TTCN has always been associated with conformance testing. In order to open the language to a wider range of testing applications in both the standards domain, and the industrial domain the specification of TTCN-3 is separated into several parts. The first part is the core language. The second part is the tabular presentation format, similar in appearance and functionality to earlier versions of TTCN. The third part is the graphical presentation format. The fourth part contains the operational semantics of the language.

The core language serves for three purposes:

- as a generalized text-based test language in its own right;
- as a standardized interchange format of TTCN test suites between TTCN tools;
- as the semantic basis (and where relevant, the syntactical basis) for various presentation formats.

### 5.3.2.3 Tools

**Testingtch TTmodeler** [TTM09] integrates the model-based system and test design and creates a direct, automated connection to the powerful test development and execution environment TTworkbench. The purpose of TTmodeler, one of the TTworkbench plug-ins, is to provide the possibility to automatically generate TTCN-3 code from a test model designed in UML/UTP. The quality of the test models and also the quality of the generated TTCN-3 test cases is assured at different levels.


**JUMBL** [JUMBL] provides several methods for the automated generation from statistical test models. Each transition in the usage model is a stimulus to the system and a path through the usage model is a sequence of stimuli. Consequently, test-case generation essentially involves automatically generating paths, i.e., a sequence of stimuli through the usage model from a beginning usage state to an ending usage state. The statistical testing method includes various ways of automated test case generation, i.e., different strategies to generate paths through the usage model. Some strategies include:

- a) Generating the minimum-cost coverage set for a usage model; in a special case when unit costs are assumed for all arcs, a collection of test cases that visit every arc in a model with the minimum number of test steps is generated
- b) Generating tests randomly based on a probability distribution
- c) Generating tests in order by weight (most or least probable paths)

Additionally, one can generate enough test cases to ensure that the so-called optimal reliability of the generated test cases lies at a desired level. One can also reduce the variance of the reliability estimation by generating more test cases. There is even the possibility to generate the test cases manually if specific test cases are to be run. In complex usage models with rarely traversed arcs, random test case generation cannot assure the coverage of the whole model, even with a large number of test cases. Therefore, the model coverage test cases should be generated first before deriving random test cases.

## 5.4 TEST IMPLEMENTATION

Abstract test cases generated from models need to be enriched (e.g. with configuration data) and adapted to the concrete SUT interfaces in order to be executed within real test campaign. Since this is not

	White Paper: V 1.6	Page : 44 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

a specific problem in the MBT area here it is only briefly addressed. Some approaches used in D-MINT are introduced in the following subsection.

### 5.4.1 Methods

**Compilers** are used to translate the abstract test case from their specific format to an executable code. The output of this step could be e.g. a conventional programming language or a specialized byte code for some engine that is able to run at the target platform. In any case, there is some adaptation needed to connect the test engine with the environment.

**Backend-scripts.** The backend concept proposes translating abstract test cases (e.g. from graphical model) into an executable script using another syntax (e.g. HTML, TCL or TTCN-3). This step requires so-called backend scripts and produces another representation of the test cases.

**Standardized Interfaces.** Executing TTCN-3 tests addresses standardized interfaces for the execution of abstract TTCN-3 statements: TTCN-3 Runtime Interface (TRI) and TTCN-3 Control Interface (TCI). A set of operations have been specified with the IDL (Interface Definition Language) as part of the TTCN-3 multipart standard. Interface operations need to be realized by special coder/decoder and test adapters that apply to the SUT interface.

**Virtual machine.** Some adaptation approaches are based on virtual machines that will run the test cases executable code directly. In such cases, it is the virtual machine instead of the individual abstract test operation that needs to be adapted to the environment. This approach is platform independent as long as the virtual machine can communicate with its target environment.


### 5.4.2 Notations

**TTCN-3** as an abstract test language requires compiling the test specification. The output of this step is the Executable Test Suite (ETS). It handles the execution or interpretation of test cases, the sequencing and matching of test events, as defined in the corresponding TTCN-3 modules. According to the TTCN-3 standard conceptually the ETS needs to be combined with a TTCN-3 Runtime System (T3RTS) and an optional Encoding/Decoding System (EDS) entity to become a so-call TTCN-3 Executable (TE).

The ETS interacts with the T3RTS entity to send, attempt to receive (or match), and log test events during test case execution, to create and remove TTCN-3 test components, as well as to handle external function calls, action operations, and timers. The T3RTS entity interacts with other entities of a TTCN-3 system via TCI and TRI, and manages ETS and EDS. A complete TTCN-3 test system consists of the TE together with a CD and TA. The above mentioned entities can base on any programming language applicable to run on the target processor platform.

**Qtronic Modeling Language.** To be able to communicate with the environment one needs to be able to send and receive variable data types. In QML, communication with the environment is done via adapters. Qtronic and adapters communicate with each other by passing generic data types called *datum*. A datum is a generic data type, which can be used to represent any other data type. The *adapter* is responsible for converting the datum into a format understood by the SUT, hence converting the abstract test case into an executable test.

**The EAST language** [Neth] is a graphical test case language to represent the call flow of the scenarios they want to test. The graphs appear similar to Specification & Description Language (SDL) diagrams

	White Paper: V 1.6	Page : 45 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

but the graphs have more functionality. *Test Independent Objects* (TIOs) are the building blocks of the graphical *Test Cases* (TC). As the name suggests, TIOs are very generic in nature and can be customized to meet a wide range of requirements. There are a variety of functions performed by placing these intelligent objects in the test path. Some of the functions include:

- Runtime access to multiple databases
- Message sequences (macros)
- Timing control
- Bearer (voice and data) functionality
- Conditional branching
- External control
- Process control (application start/ stop)
- Resource control (ports, IP addresses)
- Statements of variable assignments and operations
- Transmitting data files (XML, etc)
- User I/O control

NetHawk EAST provides a number of features that facilitate the reuse of work. Test Logic Procedures (TLPs) are the most powerful of these features. The TLP is a user defined procedure that can be called from anywhere within a TC. The TLP encapsulates a piece of call flow logic (e.g. setting up an H.323 call).


Multiple arguments can be passed to the TLP and results can be provided back to the TC. The ability to pass arguments allows for the creation of generic TLPs. A TLP can be made to behave differently by passing different values to the arguments. The argument list is the 'signature' of the TLP and that is the only component of the TLP that a user is required to know. TLPs are reusable and repeatable across all of the TCs in a Test Suite.

*Test Logic Threads* (TLT) are similar to TLPs in concept, in terms of encapsulation of logic, argument passing and reusability. The difference lies in the way TLPs and TLTs are executed. Whereas TLPs can only be executed sequentially, TLTs can be executed concurrently. TLTs are an elegant way to simulate multiple legs of a call. This is the case even if the call traverses multiple interfaces with multiple protocols in the same TC.

### 5.4.3 Tools

**TTworkbench/TTCN-3 Express** [TTWB09] is a graphical test development and execution environment based on the international standardized testing language TTCN-3. It includes the full range of features needed for test specification, execution, and analysis. Apart from text-based TTCN-3 test specification, TTworkbench also offers the option of graphic definition of test cases. Once tests have been defined in TTCN-3, TTworkbench compiles them in an executable test suite. The comprehensive test management and test execution environment feature allows users to administer, execute and analyze their tests. TTworkbench supports test automation methods which significantly reduce costs and ensure optimal quality throughout the whole test cycle.

Both meta models are part of a respective EMF model architecture, and thereby both MOF compliant. This transformation step hosts also the second, more specific validation layer. Errors detected here, depending on their criticality, are both reported in the problems view and also marked in the generated TTCN-3 code. Finally, the resulted TTCN-3 model is converted to the TTCN-3 textual representation format via a text generation component. The created TTCN-3 source code is validated, compiled and executed using TTworkbench.

	<p>White Paper: V 1.6</p>	<p>Page : 46 of 51</p>
		<p>Version: 1.6 Date : 06/11/2009</p>
		<p>Status : Final/Released Confid : Public</p>

TTCN-3 tools such as TTworkbench comprise the entities of a standardized TTCN-3 test system. Depending on the SUT interface the CD and/or TA is also provided as part of a complete test solution implemented by a tool vendor or can be generated with additional utilities. TTworkbench is a Java-based test system.

In **Qtronic**, the conversion for abstract test cases into executable test cases is done via adapters. Depending on the choice of generation method (online or offline), one has to specify a system adapter or a scripting back-end. The purpose of the adapter and the scripting back-end is to convert the abstract tests into executable tests. The conversion is performed during the test generation phase. In the approach employed in the NSN case study, the scripting back-end also is used to concretize test cases. This is done using EAST Reference Libraries containing predefined message templates which are used to refine the test cases.

The **TPT**-tests are represented as graphical test models that comprise hybrid state machines including detailed, precise signal definitions necessary for test execution. TPT test models are persisted to XML based TPT model files. For execution the graphically modeled test cases are compiled into highly optimized TPT byte code which is very compact and fast and even runs in real-time environments with a rate of less than 100 microseconds. The TPT byte code is platform independent and can be executed using the TPT virtual machine (TPT-VM) that is specially geared to run TPT byte code. Technically speaking this TPT-VM is a library written in ANSI-C and can be integrated in many test environments that handle signal based simulations or tests. Thus, the integration of the TPT-VM into an existing test environment requires the TPT-VM to be embedded into the test environment and signal data to be exchanged between the test environment and the TPT-VM at run-time – which is normally possible with little effort. For each time the TPT-VM computes the current input values (stimuli) for the SUT and observes the current SUT behavior by means of the output values of the SUT (observations). The signal traces are recorded to a compact trace file in the platform independent TPTBIN format.

**JUMBL** provides the possibility of adding test runner-specific to test model transitions and states. The scripts contain all the information for stimulating the test object and evaluating its response in the given test environment. An abstract test case is path through from the start to the exit state traversing a sequence of transitions. The scripts of the model elements traversed by the test case are concatenated to the executable test case script. JUMBL provides the definition of several test scripts in one test model (e.g. C-code and Java-code).


## 5.5 TEST EXECUTION

### 5.5.1 Methods

In **online** testing, tests are generated one by one and applied on-the-fly against the SUT. The MBT tool selects and executes test automatically based on the system model, and evaluates the responses from the system under test to check that they conform to the model. Typically, one needs an adapter to connect the SUT to the MBT tool. The purpose of the adapter is to concretize the abstract input values into values that are understood by the SUT.

In **offline** testing, the MBT tool will automatically generate test specifications directly based on a model describing the behavior of the system under test. The generated tests can be executed against the SUT either manually or with the help of a test executor. The latter helps speeding up the test execution process if one has many tests that need to be executed.

**Stress testing** exercises the system at peaks level of activity (beyond the limits of normal operation). Stress testing is particularly important for safety-critical software, but is used for all types of software.

	White Paper: V 1.6	Page : 47 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

Typically in stress testing the main focus is put on the robustness, availability, and error handling under a heavy load.

**Load testing** refers to the analysis of the expected usage of a software program by simulating multiple users accessing the program concurrently. As such, this testing is most relevant for multi-user systems. The most accurate load testing occurs with actual, rather than theoretical, results.

Load testing refers to stress testing, i.e., when the load placed on the system is raised beyond normal usage patterns, and the systems response at unusually high or peak loads is tested. The load is usually so great that error conditions are the expected result. No clear boundary exists when an activity ceases to be a load test and becomes a stress test, though. There is little agreement on what the specific goals of load testing are. The term is often used synonymously with performance testing, reliability testing, and volume testing.

**Model-in-the-Loop (MiL):** The first integration level, MiL, is based on the model of the system itself. In this platform the SUT is a functional model or implementation model that is tested in an open-loop (i.e., without any plant model in the first place) or closed-loop test with a plant model (i.e., without any physical hardware) [KHJ07]. The test purpose is basically functional testing in early development phases in simulation environments such as ML/SL/SF.

**Software-in-the-Loop (SiL):** During SiL the SUT is software tested in a closed or open-loop. The software components under test are usually implemented in C and are either hand-written or generated by code generators based on implementation models. The test purpose in SiL is mainly functional testing [KHJ07]. If the software is built for a fixed-point architecture, the required scaling is already part of the software.

**Processor-in-the-Loop (PiL):** In PiL embedded controllers are integrated into embedded devices with proprietary hardware (i.e., ECU). Testing on PiL level is similar to SiL tests, but the embedded software runs on a target board with the target processor or on a target processor emulator. Tests on PiL level are important because they can reveal faults that are caused by the target compiler or by the processor architecture. It is the last integration level which allows debugging during tests in a cheap and manageable way [KHJ07]. Therefore, the effort spent by PiL testing is worthwhile in almost all cases.

**Hardware-in-the-Loop (HiL):** When testing the embedded system on HiL level the software runs on the final ECU. However the environment around the ECU is still a simulated one. ECU and environment interact via the digital and analog electrical connectors of the ECU. The objective of testing on HiL level is to reveal faults in the low-level services of the ECU and in the I/O services [SZ06]. Additionally, acceptance tests of components delivered by the supplier are executed on the HiL level because the component itself is the integrated ECU [KHJ07]. HiL testing requires real-time behavior of the environment model to ensure that the communication with the ECU is the same as in the real application.


## 5.5.2 Notations

From the testing viewpoint there are no specific requirements or constraints on (programming) languages for test execution as long as they are executable on the target system.

## 5.5.3 Tools

The **Conformiq Qtronic** [Qtro09] tool can also be used for online testing (versions 1.x). In this mode the tool will design test cases and apply them against the SUT on-the-fly. An adapter is needed for concretizing the tests and for mediating the connection with the SUT.

**NetHawk EAST** [Neth] is a testing tool for telecommunication manufactures and operators. NetHawk EAST covers technologies including wireless, VoIP and legacy networks. A single multi-technology test

	<p>White Paper: V 1.6</p>	<p>Page : 48 of 51</p>
		<p>Version: 1.6 Date : 06/11/2009</p>
		<p>Status : Final/Released Confid : Public</p>

case enables true end-to-end testing. NetHawk EAST is used in regression, functional, load and conformance testing. NetHawk EAST tests the system under test (SUT) by simulating all the interfaces connected to the SUT. Interface simulation is implemented with state-machines that are defined in SDL like a graphical test case language. NetHawk EAST provides graphical editor for editing state-machines. With editor, customers can modify standard tests cases and develop new sophisticated testing scenarios limited only by the user's imagination and skills.

When running tests, NetHawk EAST executes test scenarios by wrapping SUT. For example, when testing eNB NetHawk EAST simulates S1, X2, and LTE Uu interfaces and simulates both control plane and user plane (with the help of supported Test Mobiles).

NetHawk EAST can be used in functional and load testing. In functional testing, NetHawk EAST can send or receive any protocol messages of the supported interfaces. It also generates and receives user plane messages. In load testing, NetHawk EAST simulates heavy load of control plane and user plane traffic. NetHawk EAST is delivered with necessary hardware that provides performance needed for heavy load testing.

In D-MINT, EAST has been used together with Qtronic for model-based testing of MSC in NSN case study. With EAST, tests can be repeated same way as many times as required thus improving test reliability and efficiency.


**Testingtech TWorkbench/TTCN-3 Express** [TTWB09] is a highly integrated test development and execution tool series for a wide range of industry domains, including the telecommunications, automotive, and financial domains, produced by Testing Technologies IST GmbH. TWorkbench supports the entire lifecycle of TTCN-3 based tests with textual and graphical editors, a TTCN-3 to Java compiler, and a test execution management environment composed of graphical tracing, debugging and reporting facilities for centralized and distributed test components. TWorkbench supports, additionally to TTCN-3, different system modeling languages such as ASN.1, IDL, WSDL and XML.

TWorkbench offers the TTCN-3 code writer besides the possibility to group and filter the error messages also Quick Fixes, which are hints on how the warnings and errors may be corrected either semi- or fully automatically. Code refactoring, an important step of the agile development process, is supported both, at the code level by functions such as organize imports and identifier renaming and also at higher level by advanced refactoring methods such as the TTCN-3 Extract Wizard and the Interface Analyze Wizard.

The TTCN-3 Extract Wizard is meant to help extracting declarations from modules belonging to one or more TTCN-3 projects. This may be used at the end of the implementation phase in order to remove unused code fragments or in order to deliver only a subset of test cases from the development branch of the test suite. The Interface Analyze refactoring wizard presents the possibility to keep track of and to react to interface changes between two versions of a given TTCN-3 project. It allows the user to record these changes and apply them to newer versions of the projects. To ease creation and modification of TTCN-3 templates a Template Wizard has been integrated into the Core Language Editor. It provides the possibility to easily specify and validate complex data structures on the basis of their type constraints using a simple graphical interface.

**iXtronics CAMEL-View** [iXtronics] is an environment for the development of mechatronic systems and offers a variety of functions to fully support the development cycle of innovative systems. This includes the modeling of the physical system based on its topology as well as the implementation of the more functional control concepts. The implementation of the model is supported by means of a graphical user interface and is based on a fully textual and readable syntax which is checked during import resp. modification of the model.

The modeling based on the physical characteristics of the system implies that a mathematical

	White Paper: V 1.6	Page : 49 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

representation of the system for simulation and analysis purposes is not directly available. This mathematical representation, based on non-linear differential equations, is generated automatically. For multi body systems the automatic generation supports different generation algorithms (dynamic -coupling, minimal formalism and recursive formalism). To allow fast analysis of the system and furthermore, to support real-time capability, the generated code is further reduced and optimized.

The simulation is performed with different kind of integration algorithms which can be chosen by the user. To ensure numerical stability the stability range of the chosen integrator can be plotted and checked against the Eigen values of the system.

Due to the textual description of the model components, arbitrary components of the model can be generated by third party tools, e.g., to generate an excitation block as part of the model to be evaluated. Furthermore, CAMEL-View supplies a RMI-interface which allows other tools to control CAMEL-View remotely. The RMI-Interface allows to one change parameters, to read outputs and to control the generation of the mathematical model as well as its simulation.

With the physical interfaces to the real world system (DAC, ADC, CAN), CAMEL-View TestRig supports the testing of real components (e.g., the ABB soft starter). To do so, the simulated model must guarantee hard real-time which is continuously checked during evaluation of the simulation model. Furthermore, the user can check the required time for each evaluation step and compare it with the integration step size.


Full control of the SUT is given by means of instrumentation components in the analysis window which provides plot facilities as well as means to change parameter. Hence, the user has a visual impression of the test which is just going on and its results.

Within D-MINT CAMEL-View provided the interface to the ABB soft starter and all required model components to stimulate the real system as well as reading significant outputs and parameters of the systems. This test environment is utilized by TPT (PikeTec) and TTworkbench (Testing Technologies).

**Elvior MessageMagic** [EMM] is a TTCN-3 test development and execution platform. It can be used for testing software or hardware components in a wide range of industry sectors. Candidate systems for testing with MessageMagic can be found from the industry sectors where TTCN-3 is widely used as an accepted generic test language - communication, transport and automotive, military and defense.

MessageMagic is ideal for incremental project development. It can be used for testing of an individual task or process. One can continue with the integration tests of a subsystem consisting of processes and later to integrate those subsystems and test them together as a whole software subsystem. It is even possible to continue to perform SW/HW integration tests where MessageMagic can be used to test SW running in target HW.

MessageMagic simplifies and accelerates product development, with all the benefits that derive from shorter development cycles. MessageMagic relies on TTCN-3 architecture and supports TTCN-3 core language. MessageMagic is used for executing generated tests in ELIKO street lighting controller and in Trimek/Datapixel case studies. The quality of the TTCN-3 tests can be ensured by several important features. Setup of test environments is intuitive and user-friendly with MessageMagic reducing tester's time to setup the test environment. MessageMagic includes graphical IDE for managing TTCN-3 testing projects. MessageMagic has user-friendly and intuitive environment to manage the testware. Messages are logged in a user-friendly format by using field names with values and symbols instead of numeric constants. This is an extremely important and productive feature. It completely eliminates the need to deal with low-level message representations, such as unformatted strings of hexadecimal numbers.

	White Paper: V 1.6	Page : 50 of 51
		Version: 1.6 Date : 06/11/2009
		Status : Final/Released Confid : Public

## 5.6 TEST REPORTING

### 5.6.1 Methods

**Log-based analysis** is the means to analyze the output of a test run. The analysis is based on log files that are created after the test execution is finished. The log files contain information of what test cases, requirements, methods, etc. have been executed/covered during the test execution phase. From these test logs one can extract information related to number of failed/passed test cases and statistics on how different test requirements have been covered. More information can be found in [ATL09ATL09].

**Back-tracing of requirements** is a technique for tracing requirements from failed test cases back to system or test models. This way one can see which requirements failed during testing and to what model elements they are linked. Ultimately, since requirements are linked to model elements, it facilitates the identification of which functionalities of *the real system* are not in sync with the *model*, and hence with the requirements. This is only possible if requirements are traced throughout the whole testing process. That is, requirements are traced to the system model, to the test model, and to the test cases. If requirements are traced to model elements, and further to test specifications, it makes possible to trace back failed requirements to system models. More information can be found in [ATL09].

**Root cause analysis.** Model-based monitoring is a specific form of model-based testing which is applicable to both the pre-production and post-production, i.e., online, phases of a system. Online testing indicates that a connection is made to a system under test and it is tested dynamically during execution. Model-based monitoring assists in locating the fundamental reason (a.k.a. the root cause) of an error.

The most distinctive differentiations between model-based testing and model-based monitoring are the ensuing: 1) model-based monitoring may perform certain reactions, e.g., failure elimination or mitigation, to particular inputs, outputs and states, and 2) model-based monitoring can be utilized in post-production, i.e., online systems, to locate errors and predicaments that incur in the system.


Model-based monitoring software components may be applied either: 1) solely to the pre-production phases of systems development, i.e., they are eradicated before the system is launched, or 2) the post-production, i.e., online, phases of the system. In either of these phases, the model-based monitoring can discharge the actions of 1) collecting the monitored information, or 2) collecting and responding to the monitored information. In the D-MINT project, the model-based monitoring can discharge the actions of 1) collecting the monitored information, or 2) collecting and responding to the monitored information.

### 5.6.2 Notations

In the NSN case study, the Qtronic tool generates test logs for online testing in the HTML format and the EAST TestRunner generates the logs files in textual format. A combination of **Python** scripts and **OCL** queries is used to trace back test cases to the UML models of the SUT.

### 5.6.3 Tools

**TTworkbench/TTCN-3 Express** supports the generation of test reports in HTML, PDF, Excel, or Word format. The Excel format is an XML file that can be opened with Microsoft Office Excel 2003 and later versions as well as with OpenOffice 2.0. The Word format is also an XML file that can be opened with Microsoft Office Word 2003 and later versions. An image can be generated and saved in the test report directory for each test case. In the test report, the test case name will be linked to the correspondent image file.

	<p>White Paper: V 1.6</p>	<p>Page : 51 of 51</p>
		<p>Version: 1.6 Date : 06/11/2009</p>
		<p>Status : Final/Released Confid : Public</p>

**Qtronic** offers the possibility to log the execution of test cases by specifying a scripting back-end. A scripting back-end is a plug-in that is connected to Qtronic and produces a report containing information of the test run. The test report contains information about the test cases, failed/passed requirements, tested methods, etc.

In **EAST**, every execution of a test script can be saved in a log file. The log file is a simple text file containing the output of the test script execution. In the NSN case study, the **MATERA** framework is used to analyze the Qtronic or EAST test logs in order trace-back requirements to the system models of the SUT using a **script** written in the Python language. The script produces a series of OCL queries that are used to interrogate the system models in the **MagicDraw** model editors. Upon running the produced queries, the parts of the models pertaining to the selected requirements are highlighted.

**Root Cause Analyzer.** The realization of the model-based monitoring technology and methodology is modeled as the Root Cause Analyzer in the Nokia Siemens Networks Case Study. The logs created by the executor component and the SUT adaptor are employed by the Root Cause Analyzer. Once an error incurs, either online or offline, the Root Cause Analyzer endeavors to evaluate the reason for the error.

The inputs utilized for distinguishing the error are:

- the models from the Editor, and
- the logs provided by the Executor and the SUT Adaptor.

From these models and logs, the Root Cause Analyzer attempts to ascertain the ultimate and basic cause of the error. Currently, the error location has proved to be a tedious and arduous task. The utilization of the Root Cause Analyzer would abate the time and cost to discover the actual location of an error.

The outputs of the Root Cause Analyzer are:

- Verbal description of the detected failure
- Detailed raw data regarding the failure (e.g. message contents)
- Level of the failure's seriousness (as perceived by the RCA)
- Suspected source(s) of the fault

The actual root cause analysis process on received input is performed by the Logical Root Cause Analyzer subsystem. During analysis, events parsed from received test logs are compared to the expected SUT behavior described in the model. From these differences between test output and modeled behavior, the Logical Root Cause Analyzer endeavors to deduct the causes of detected failures. The analysis process is managed using a rule-based system, which is used to define the aspects of test activity to monitor. In the context of the telecommunications use case, the RCA may for example be instructed to analyze certain sequences of messages between the Test Executor and the SUT. This is done in order to ensure that e.g. field values and the order of messages remain constant with modeled behavior.

The analysis reports produced as the output of the Root Cause Analyzer are implemented in a format which is both machine and human readable. This enables possible automated use of monitoring results by software, as well as providing readable information to the test designing personnel. The reports provide both factual and subjective data regarding failures and their causes detected during the analysis process. Factual data contains raw information such as exact message contents, while subjective data consists of deductions and suggestions made by the Root Cause Analyzer itself.

**JUMBL** provides a detailed statistical analysis and report of the test results. Failures are mapped to the transitions and stimuli that caused it. Based on test cases and the failure distribution the system reliability is estimated. Different reliability metrics are calculated which can be used for assessing the current system quality and the status of the test project. All reports are generated as HTML-pages.